JSF 1.2 Study Notes

by

Ivan A Krizsan

Version: May 16, 2008

Copyright 2008 Ivan A Krizsan. All Rights Reserved.

Table of Contents

Table of Contents....2
Purpose 5
Licensing 5
Disclaimers 5
Environment....5
What is JSF?....6
Extendibility....6
- Custom User-Interface Widgets....6
- Custom Display Technologies....6

JSF and MVC....7
Parts of JSF....8
- User Interface Components....8
  - JSF HTML Tags....10
  - Facets....11
  - JSF Core Tags....13
    - The <f:verbatim> Tag....14
- Managed Beans and Backing Beans....15
  - Implementation....16
  - Configuration....17
  - Use....17
  - Initialization....18
  - Managed Lists and Maps....18
  - Managed Bean Properties....20
- Validators....22
  - Validator Classes....22
  - Validator Methods....24
- Messages....25
- Converters....27
  - Attaching Converters....28
  - Converter Tags....29
  - Converter Classes....30
  - Converter Messages....31
- Renderers....32
- Events and Listeners....33
  - Action Events....33
    - The actionListener Attribute....33
    - The <f:actionListener> Tag....34
    - The <f:setPropertyActionListener> Tag....35
    - Invocation Order....35
  - Application Actions....36
  - Value Change Events....37
    - The valueChangeListener Attribute....37
    - The <f:valueChangeListener> Tag....37
  - Phase Events....38
    - Phase Listener....38
    - Registering a Phase Listener....38

Request Processing Life Cycle....39
- Restore View....41

Apply Request Values....42
Process Validations....43
Update Model Values....44
Invoke Application....44
Render Response....45
Navigation....46
Localization and Internationalization....48
Resource Bundles....49
Accessing Resource Bundles....50
Configuring Supported Locales....50
Retrieving the Message....51
Controlling the Locale....51
Browser's Current Locale....51
JSF's Current Locale....51
Retrieving Locale Configuration....52
Core Tags and User Interface Components in Detail....53
Attributes and Custom Attributes....53
Making Selections....55
Single Checkboxes....55
Selecting from a List....55
Hardcoded List Items....55
Fallback List Items....56
Multiple Items....57
Grouping Multiple Items....58
Glossary....59
Appendix 1 – Events and Request Processing Life Cycle Sample Program....61
Web Page....61
Managed Bean Class....62
JSF Configuration File....64
Web Application Deployment Descriptor....64
Analysis....65
Earlier Validation....67
Skipping Phases....69
Bailing Out....71
Appendix 2 – UI Component Tree Navigation....73
Web Page....73
Managed Bean Class....75
JSF Configuration File....77
Web Application Deployment Descriptor....77
Analysis....78
Appendix 3 – Navigation Sample Program....81
Navigation Schema....81
Java Script....82
Web Pages....82
cancelled.jspx....82
completed.jspx....83
input_address.jspx....84
input_firstname.jspx....85
input_lastname.jspx....86
introduction.jspx....87

Managed Bean Classes......88
BaseController......88
CancelledController......89
CompletedController......90
InputAddressController......91
InputFirstNameController......91
InputLastNameController......92
IntroductionController......92
PersonBean......93
Web Application Deployment Descriptor......94
JSF Configuration Files......95
faces-config.xml......95
faces-beans.xml......96
faces-navigation.xml......97
Analysis......98
Navigation......98
Splitting the JSF Configuration File......99
Giving Focus to Input Components......100
Appendix 4 – Internationalization of a Web Page......101
Web Page Before......101
Creating a Messages Property File......102
JSF Configuration File......103
Web Application Deployment Descriptor......104
Controller......105
Controller Analysis......106
Web Page After......107
Appendix 5 - Building a WAR File......108
Using NetBeans......109
Including Libraries......109
Including Additional Content......110
Building the WAR File......111
Using Eclipse......112
Register JSF Libraries......112
Creating the Project......114
Building the WAR File......116

Purpose

This document contains the notes I made when studying JavaServer Faces 1.2. It is meant to given an introduction to JavaServer Faces 1.2 for someone who is familiar with JSP and servlet programming.
It contains my interpretation of the material studied and some code I wrote to help me understand the different concepts. I haven't covered every single aspect of JSF, I neither have the time nor the expertise.

Licensing

This document is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 License.

Disclaimers

Though I have done my best to avoid it, this document might contain errors. I cannot be held responsible for any effects caused, directly or indirectly, by the information in this document – you are using it on your own risk.
Submitting any suggestions, or similar, the information submitted becomes my property and you give me the right to use the information in whatever way I find suitable, without compensating you in any way.
All trademarks in this document are properties of their respective owner and do not imply endorsement of any kind.
This document has been written in my spare time and has no connection whatsoever with my employer.

Environment

All the example code in this document were developed using JSF 1.2 and an integrated development environment. NetBeans has JSF libraries “built-in”, while Eclipse needs to be configured – see the appendix 5 on how to set up Eclipse for JSF development.
All examples have been run on Tomcat 6.

What is JSF?

JSF is a server-side framework for developing web-based user interfaces using components.
JSF defines the following things to aid in the development of web applications:

- A component architecture.
  Defines how user-interface widgets are created. Not only does the standard components adhere to the component architecture, but it also allows for custom component development.
- A set of user-interface widgets.
  Pre-defined standard components like buttons, text-fields etc.
- An application infrastructure.
  Enforces development using the MVC (model-view-controller) design pattern.

Technically, JSF is implemented using servlet technology and, as one of the choices available for display technology, HTML, using custom tags.

Extendibility

As mentioned earlier, JSF may be extended in the following ways:

- Custom user-interface widgets.
- Custom display technologies.

Custom User-Interface Widgets

The development of custom user-interface widgets is outside the scope of this document. This subject may be discussed in a separate, future, document.

Custom Display Technologies

In order to enable support for a custom display technology for standard JSF components, the following has to be done:

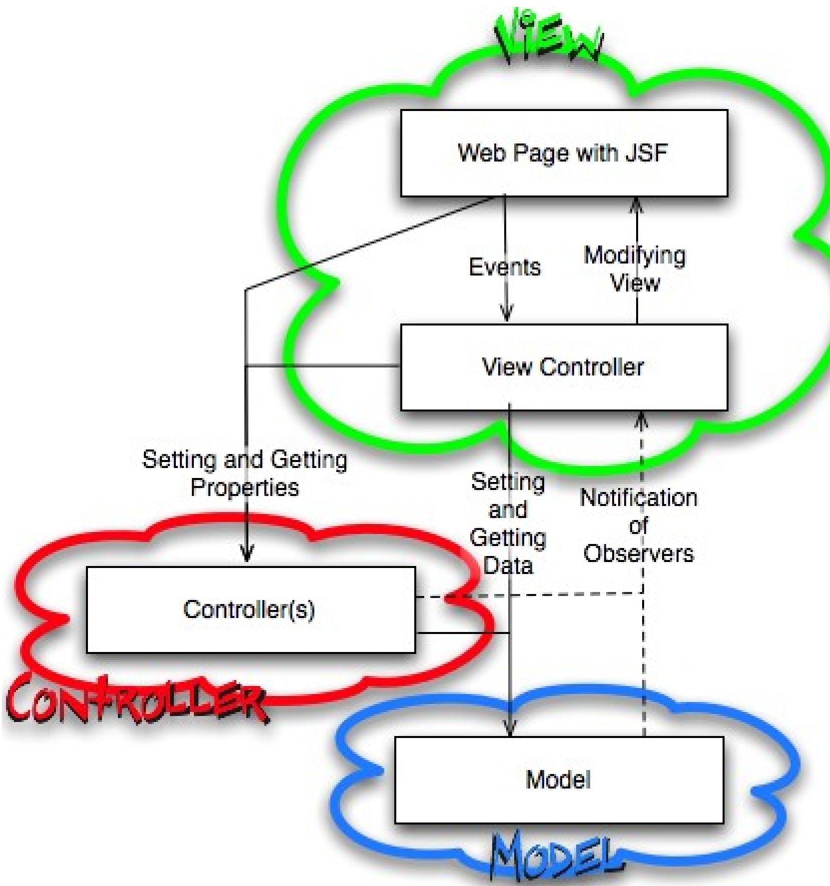
- If the custom display technology require additional data for a UI component, create a subclass for that component and add the data in question.
- Implement renderer classes for the UI components.
- Implement a renderer kit for the new display technology.

Please refer to the Renderers section below for more information on the JSF rendering model.

JSF and MVC

From a developer's point of view, the different parts of the MVC design pattern corresponds to the following JSF entities:

- The view is realized by a number of JSP pages and, as needed, backing beans to implement event listeners. The function of an event listener class is to forward events to a controller, using a display-technology neutral way.
  Event listeners are placed in the view in order to isolate the controllers from JSF-specific view technology, for instance *javax.faces.event.ActionEvent*.
- The controllers are implemented by JavaBeans (managed beans).
- The model may consist of plain Java objects, EJBs etc.



Model, view and controller parts of a JSF application and their interactions.

This way of implementing the MVC design pattern results in controllers that are independent of the view technology, which can be re-used if the view is exchanged. The view-controllers, belonging in the view, however, does not share this property, but have to be re-written for each view technology used.
On a technical level, both view-controllers and the regular controllers can be implemented as managed beans.

Parts of JSF

This section describes the fundamental parts of JSF:

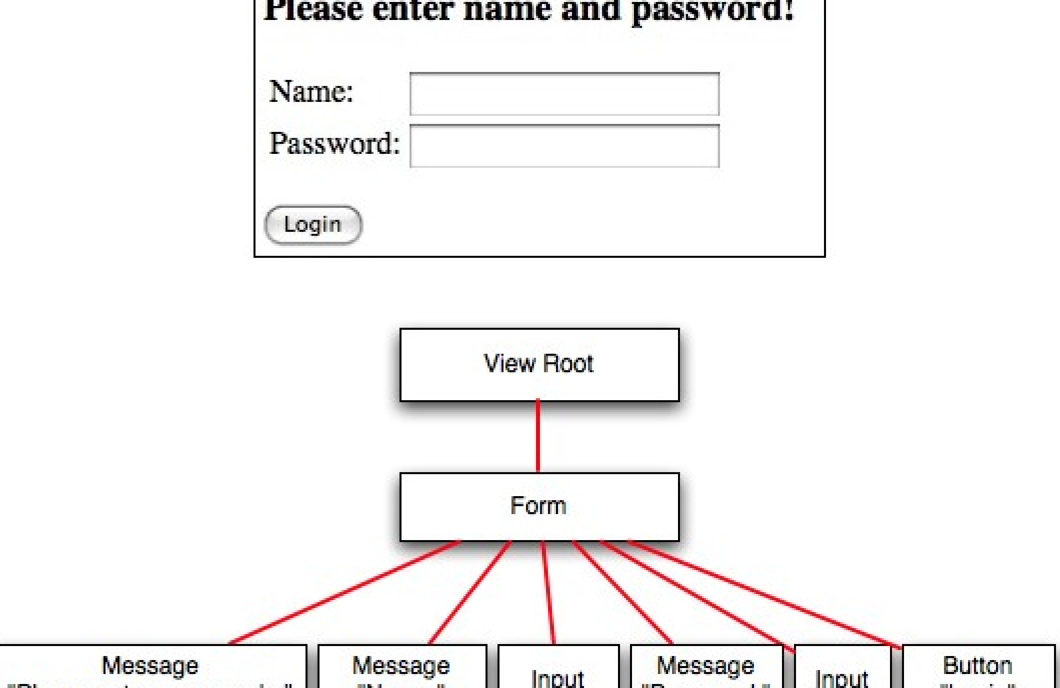
- User Interface Components
- Validators
- Messages
- Converters
- Renderers
- Managed Beans and Backing Beans
- Events and Listeners

User Interface Components

Reference: JSF Specification 3.1, 4.1, 9.5

This section contains a basic introduction of user interface components of JSF. For more detail and examples, please refer to Core Tags and User Interface Components in Detail later in the document!

JSF user interface components are the basic building blocks for creating a JSF user interface. They are configurable and reusable, with complexity ranging from simple, such as buttons or text fields, to high, such as components built by combining simpler components.
Similar to Swing and many other user interface frameworks, the JSF user interface components maintain a state and are arranged in a tree structure.



A simple JSF view and the corresponding user interface component tree.

The JSF code that creates the above web-page is shown in the listing below. Note that parts not significant to the discussion of JSF user interface components have been left out.

```
<f:view>
    <html xmlns="http://www.w3.org/1999/xhtml">
        <head>
            <title>Login Page</title>
        </head>
        <body>
            <h:form>
                <h3>
                    <h:outputText value="Please enter name and password!"/>
                </h3>
                <table>
                    <tr>
                        <td>
                            <h:outputText value="Name:"/>
                        </td>
                        <td>
                            <h:inputText value="#{user.name}" required="true"/>
                        </td>
                    </tr>
                    <tr>
                        <td>
                            <h:outputText value="Password:"/>
                        </td>
                        <td>
                            <h:inputSecret value="#{usr.psw}" required="true"/>
                        </td>
                    </tr>
                </table>
                <p>
                    <h:commandButton value="Login" action="#{user.check}"/>
                </p>
            </h:form>
        </body>
    </html>
</f:view>
```

Source-code for the simple JSF web-page displaying a login form.

The JSF components are, unlike Swing components, not responsible for drawing themselves. This duty is delegated to a renderer, see the Renderers section.

Every UI component in the tree hierarchy representing a view (typically a web-page) has a component identifier. If the identifier was not supplied by the programmer, then JSF automatically assigned an identifier to the component. Certain components, like data tables (tag <h:dataTable>) and forms (tag <h:form>) creates a naming container. All component identifier within a naming container must be unique.
Using component identifiers and methods available in the *javax.faces.component.UIComponent* class, components in the tree hierarchy can be located.
See appendix 2 for a detailed discussion on component tree navigation and a complete example program.

JSF HTML Tags

The following table lists the JSF HTML tags, which have the prefix h. References in the Description section refers to code example(s) where the tag is used.

JSF Tag	Corresponding HTML Tag	Description
h:form	<form>	Form for user input.
h:inputText		Text-field accepting a single line of text. Text is displayed in clear text.
h:inputTextarea	<textarea>	Text-field accepting multiple lines of text. Text is displayed in clear text.
h:inputSecret	<input>	Text-field accepting a single line of text. Text is not visible, but replaced by dots.
h:inputHidden	<input>	Invisible text-field accepting a single line of text. Typically used to pass variables from page to page.
h:outputLabel	<label>	Associates a text label with a control so that the control is toggled when the label is clicked.
h:outputLink	<a>	Defines an anchor, which can be used either to create a link to another document or to create a bookmark inside a document.
h:outputFormat	[none]	Text output allowing for text with parameters.
h:outputText	[none] or <span>, if style or styleClass specified.	Text output of a single line of text.
h:commandButton	<input>	Defines a button represented by either an image or a regular button (with text). See the Action Events section!
h:panelGroup	<div> or <span>	Group multiple components. Typical use in facets, where only one UI component can be nested.
h:dataTable	<table>	Defines a table.
h:column	[none]	Defines a column in a dataTable, with optional header and footer.
h:commandLink	<a>	HTML link.
h:message	[none]	Displays the last message of a component. See the Messages section!
h:messages	[none]	Displays all messages of a component in either a table or a list. See the Messages section!
h:graphicImage	<img>	Displays an image.
h:selectOneListbox	<select>	Displays a list or a popup-menu, depending on number of items to show, allowing selection of one single item.
h:selectOneMenu	<select>	Displays a popup-menu allowing selection of one single item. See Making Selections section!
h:selectOneRadio	<select> in <table>	Displays a number of radio buttons in a table and allows selection of one of them.
h:selectBooleanCheckbox	<input>	Displays a check box. Note that the label has to be displayed using h:outputText or similar. See Making Selections section!

Table continued on next page.

JSF Tag	Corresponding HTML Tag	Description
h:selectManyCheckbox	<input> in <table>	Displays a number of check boxes in a table with labels and allows for independent checking of each.
h:selectManyListbox	<select> and <option>	Displays a list allowing selection of multiple items. See Making Selections section!
h:selectManyMenu	<select> and <option>	Displays a menu (in reality a list) allowing selection of multiple items. See Making Selections section!
h:panelGrid	<table>	Displays a table containing a number of components.

Facets

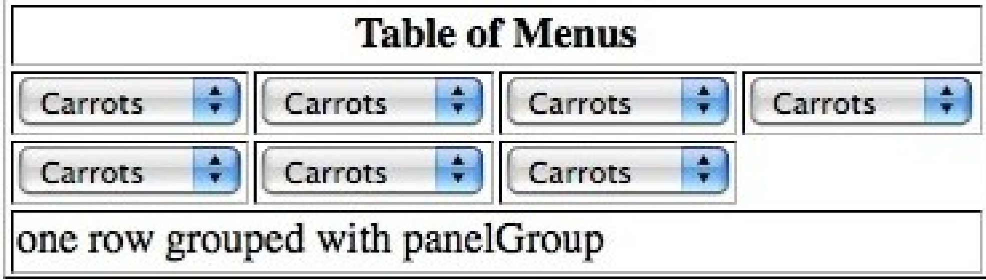
Reference: JSF Specification 3.1.9, 9.2.7

Apart from being a child of a parent, a UI component can be a facet of another UI component. This is the kind of relationship headers and footers of tables have to the table, where it is more natural that the contents of the table is children to the table itself.
Facets are not considered to be children of their parents and each facet relation has an unique name.

```
<h:panelGrid columns="4" footerClass="subtitle" headerClass="subtitlebig"
styleClass="medium"
                      columnClasses="subtitle,medium" border="1">
    <f:facet name="header">
        <h:outputText value="Table of Menus"/>
    </f:facet>
    ...
    <f:facet name="footer">
        <h:panelGroup>
            <h:outputText value="one row"/>
            <h:outputText value=" "  />
            <h:outputText value="grouped with panelGroup" />
        </h:panelGroup>
    </f:facet>
</h:panelGrid>
```

Example of facet usage: A header and footer of a table.

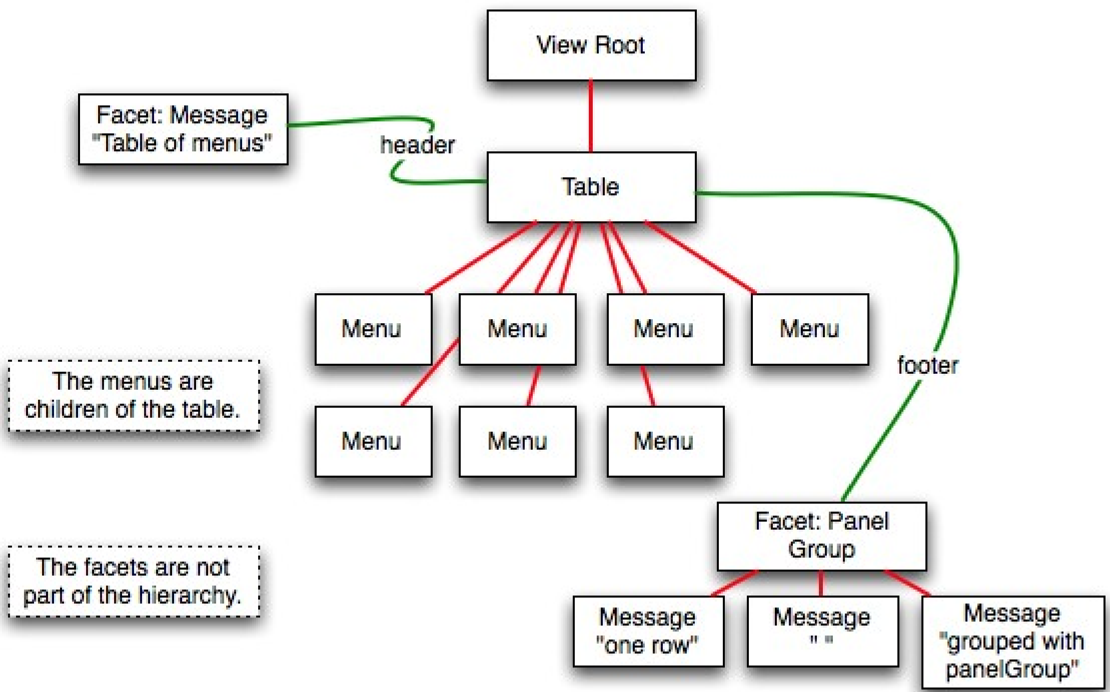
The result looks like this:



Result of using facets: A table with header and footer.

Looking at the source above we see a technique commonly used to overcome the limitation of a facet only being able to contain a single UI component (due to facets being stored in a *Map<String, UIComponent>* in the parent component). The footer of the table is made up of three output text components, grouped using a panel group.

The tree hierarchy for the above table looks like this:



Showing how facets are related to the JSF user interface component tree structure.
The example shows a table with facets for header and footer.

JSF Core Tags

Reference: JSF Specification 9.4

The JSF core tags provide core actions that are independent of the particular renderer used. These tags have been organized in a separate tag library that usually has the prefix f.

JSF Tag	Description
f:actionListener	Register an action listener on the parent UI component. See the Action Events section!
f:attribute	Add an attribute on the parent UI component. See the Attributes and Custom Attributes section!
f:convertDateTime	Register a date-time converter on the parent UI component. See the Converters section!
f:convertNumber	Register a number converter on the parent UI component. See the Converters section!
f:converter	Register a converter having specified id on the parent UI component. See the Converters section!
f:facet	Register a facet having the specified name on the parent UI component. See the Facets section!
f:loadBundle	Loads a localized resource bundle and make it available as a request-scoped map. See the Localization section!
f:param	Adds a parameter to the parent UI component. Used, for instance, to insert values into strings with parameters or append parameters to hyperlinks.
f:phaseListener	Register a phase listener on the view root. See the Phase Events section!
f:selectItem	Adds one child to the parent UI component. Parent component usually provides multiple items for selection. See Making Selections section!
f:selectItems	Adds multiple children to the parent UI component. Parent component usually provides multiple items for selection. See Making Selections section!
f:setPropertyActionListener	Register an action listener that, when the page is submitted, will retrieve a value from a value expression and store it in another value expression. See the Action Events section!
f:subview	Used to create a naming container, for instance when including other pages or fragments. See appendix 2!
f:validateDoubleRange	Adds a range validator for double values to the parent component. See the Validators section!
f:validateLength	Adds a string length validator to the parent component. See the Validators section!
f:validateLongRange	Adds a range validator for long values to the parent component. See the Validators section!

Table continued on next page.

JSF Tag	Description
f:validator	Adds a validator with specified id to the parent component. See the Validators section!
f:valueChangeListener	Adds a value change listener to the parent component. See Value Change Events section!
f:verbatim	Wraps non-JSF data in a JSF component that is added to the parent component. See The <f:verbatim> Tag section! below!
f:view	Container for all JSF tags used on a page.

The <f:verbatim> Tag

In older versions of JavaServer Faces, HTML and non-JSF tags would not be properly rendered if made child of a JSF UI component. If you are using JSF version 1.2 and JSP 2.1, then there is no longer any need to use the <f:verbatim> tag.
As an example, the following code fragment, when used with earlier versions of JSF, would cause the text “Page 2” to be erroneously rendered.

```
<h:outputLink value="page2.html" title="Go to page2">
    <h3>Page 2</h3>
</h:outputLink>
```

Code fragment that require the <f:verbatim> tag to work correctly with older, pre 1.2, versions of JSF.

Corrected, it would look like this:

```
<h:outputLink value="page2.html" title="Go to page2">
    <f:verbatim>
        <h3>Page 2</h3>
    </f:verbatim>
</h:outputLink>
```

Code fragment that will work correctly with older, pre 1.2, versions of JSF.

Managed Beans and Backing Beans

Reference: JSF Specification 5.3, 5.4, Java EE 5 Tutorial chapter 9 “Backing Beans”.
The managed bean facility of JSF allows plain Java object beans to be exposed to EL expressions.
Managed beans in JSF are plain Java objects that can have one or more of the following functions:

1. Retain properties that are accessible from JSF web pages.
2. Listen for events produced by JSF web pages.
3. Validate user input.
4. Manipulate UI component server-side instances.
5. View selection.
6. Hold model data.
7. Act as a façade for an external service.

Number one and six is not necessarily overlapping; some properties may be related to the way data is presented to the user, for instance the format of a date, and have nothing to do with the model data.
Functions one to five are traditionally tasks performed by a controller in the model-view-controller design pattern, while six is a duty of model objects. A controller is always a managed bean but a managed bean is not necessarily a controller.
Please refer to the section JSF and MVC for a further discussion on on how to implement MVC when using JSF!

Implementation

A managed bean is implemented as a Java class. This class does not need to inherit from any other class, nor does it need to implement any particular interface. In its simplest form, a managed bean stores a number of properties, for instance first and last name of a person.

```
package com.ivan;
public class PersonBean
{
    /* Instance variable(s): */
    private String mFirstName;
    private String mLastName;

    @PostConstruct
    public void initialize()
    {
        // Place bean initialization code here.
    }

    @PreDestroy
    public void cleanup()
    {
        // Place clean-up code here.
    }

    public String getFirstName()
    {
        return mFirstName;
    }

    public void setFirstName(final String inFirstName)
    {
        mFirstName = inFirstName;
    }

    public String getLastName()
    {
        return mLastName;
    }

    public void setLastName(final String inLastName)
    {
        mLastName = inLastName;
    }
}
```

A simple managed bean that stores a person's name.

Note that in the above example of a managed bean, two annotations has been used in the bean; @PostConstruct and @PreDestroy. The following annotations are available for use in JSF managed beans.

- @Resource
- @Resources
- @EJB
- @EJBs
- @WebServiceRef
- @WebServiceRefs
- @PersistenceContext
- @PersistenceContexts
- @PersistenceUnit
- @PersistenceUnits
- @PreDestroy
- @PostConstruct

Please refer to Java EE and/or JSF Specification documents for details on these annotations.

Configuration

In order to make the bean accessible from a JSF web-page, it must be configured in the faces-config.xml file, in the web application.

```
<faces-config ...>
    ...
    <managed-bean>
        <description>My first managed bean</description>
        <managed-bean-name>person1</managed-bean-name>
        <managed-bean-class>com.ivan.PersonBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
    ...
</faces-config>
```

Configuration of the simple managed bean in the faces-config.xml file.

The basic elements that need to be specified when configuring a managed bean are:

- <managed-bean-name>
  Specifies the name which is used to access the managed bean in the JSF web page.
- <managed-bean-class>
  Specifies the class that implements the managed bean. This can be a user created class or a class that implements *java.util.List* or *java.util.Map*.
- <managed-bean-scope>
  The scope in which the managed bean will be stored. Possible values are request, session, application and none. The "none" scope is useful when the bean only is to be referenced by other managed beans and not from any JSF web page. See below for an example!

Use

Having implemented the backing bean and configured it in the faces-config.xml file, it can be used from a JSF web page.

```
...
<h:outputText value="First name: #{person1.firstName}"/>
<h:outputText value="Last name: #{person1.lastName}"/>
...
```

Code fragment showing how to retrieve first and last names from the managed bean using JSF tags.

Note that when accessing properties of managed beans from JSF tags, the "#" has to be used like in the example above. When accessing the same properties from ordinary JSP or JSTL tags, the "$" has to be used, like in the example below.

```
    <c:out value="First name: ${person1.firstName}"/>
```

Code fragment showing managed bean property access from JSP or JSTL tags.

Initialization

Managed beans can not only be configured in the faces-config.xml file, they can also be initialized. Setting default first and last names in the managed bean in the above example is as simple as adding a few lines to the faces-config.xml file.

```
...
<managed-bean>
    <managed-bean-name>person1</managed-bean-name>
    <managed-bean-class>com.ivan.PersonBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>

    <managed-property>
        <property-name>firstName</property-name>
        <value>Knegert</value>
    </managed-property>
    <managed-property>
        <property-name>lastName</property-name>
        <value>Edvaldsson</value>
    </managed-property>
</managed-bean>
...
```

Initializing a managed bean in the faces-config.xml file.

Managed Lists and Maps

Managed beans that are of the type *java.util.List* or *java.util.Map* may also be configured and initialized using the above techniques.

```
...
<managed-bean>
    <description>Managed bean that is a list</description>
    <managed-bean-name>managedList</managed-bean-name>
    <managed-bean-class>java.util.ArrayList</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <list-entries>
        <value-class>java.lang.String</value-class>
        <value>List Entry One</value>
        <value>List Entry Two</value>
        <value>List Entry Three</value>
        <value>List Entry Four</value>
    </list-entries>
</managed-bean>
...
```

Configuring and initializing a managed bean that is a list in the faces-config.xml file.

The managed list bean can be accessed from the JSF web page using, for instance, JSTL tags.

```
<c:forEach items="${managedList}" var="o" varStatus="i">
    <p>${i.index}: ${o}</p>
</c:forEach>
```

Listing the contents of the managed list bean from a web page.

A managed bean that is a map may be configured and initialized in a similar way.

```
...
<managed-bean>
    <description>Managed bean that is a map</description>
    <managed-bean-name>managedMap</managed-bean-name>
    <managed-bean-class>java.util.HashMap</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <map-entries>
        <key-class>java.lang.String</key-class>
        <value-class>java.lang.Integer</value-class>
        <map-entry>
            <key>One</key>
            <value>1</value>
        </map-entry>
        <map-entry>
            <key>Two</key>
            <value>2</value>
        </map-entry>
        <map-entry>
            <key>Three</key>
            <value>3</value>
        </map-entry>
        <map-entry>
            <key>Four</key>
            <value>4</value>
        </map-entry>
    </map-entries>
</managed-bean>
....
```

Configuring and initializing a managed bean that is a map in the faces-config.xml file.

The contents of the map can be accessed from a JSF web page using, for instance, JSTL tags.

```
<c:forEach items="${managedMap}" var="k" varStatus="i">
    <p>${i.index}: Key=${k.key}, Value=${managedMap[k.key]}</p>
</c:forEach>
```

Listing the contents of the managed map bean from a web page.

Managed Bean Properties

Managed beans can be configured, initialized and made properties of other managed beans.
Assume we have the following managed bean:

```
package com.ivan;

public class MyBean
{
    /* Constant(s): */

    /* Instance variable(s): */
    private int mInteger;
    private Integer mIntObj;
    private String mString;
    private List<Integer> mIntObjList;
    private Map<Integer, String> mMap;
    private List<PersonBean> mPersons;

    // Getter and setter method implementations excluded.
}
```

Managed bean whose properties are to be initialized.

The different properties of the above managed bean can be initialized in the faces-config.xml file in the following manner:

```
<!--
    Assume that two managed beans of the type com.ivan.PersonBean
    with the names person1 and person2 has already been configured
    and initialized.
-->
<managed-bean>
    <managed-bean-name>myBean</managed-bean-name>
    <managed-bean-class>com.ivan.MyBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>

    <managed-property>
        <description>A list holding person beans</description>
        <property-name>persons</property-name>
        <list-entries>
            <value-class>com.ivan.PersonBean</value-class>
            <value>#{person1}</value>
            <value>#{person2}</value>
        </list-entries>
    </managed-property>

    <managed-property>
        <description>A map holding person beans</description>
        <property-name>map</property-name>
        <map-entries>
            <key-class>java.lang.String</key-class>
            <value-class>com.ivan.PersonBean</value-class>
            <map-entry>
                <key>person1</key>
                <value>#{person1}</value>
            </map-entry>
            <map-entry>
                <key>person2</key>
                <value>#{person2}</value>
            </map-entry>
        </map-entries>
    </managed-property>

    <managed-property>
        <description>An integer in a wrapper class</description>
        <property-name>intObj</property-name>
        <property-class>java.lang.Integer</property-class>
        <value>2345</value>
    </managed-property>

    <managed-property>
        <description>A list of intergers in wrapper objects</description>
        <property-name>intObjList</property-name>
        <list-entries>
            <value-class>java.lang.Integer</value-class>
            <value>3</value>
            <value>2</value>
            <value>1</value>
            <value>0</value>
        </list-entries>
    </managed-property>

    <managed-property>
        <description>A primitive integer</description>
        <property-name>integer</property-name>
        <value>123</value>
    </managed-property>

    <managed-property>
        <description>A string</description>
        <property-name>string</property-name>
        <value>This is the last example!</value>
    </managed-property>
</managed-bean>
...
```

Initialization of different kinds of properties of a managed bean.

As before, managed beans that only are to be properties of other managed beans, and not directly used from web page(s), should use the <managed-bean-scope> none.

Validators

Reference: JSF Specification 3.5, 4.1, 9.4.14-9.4.17
Validators are pluggable support classes or methods that ensures that the value of a user-modifiable user interface component conforms to certain rules. For instance, if the user is asked to enter the number of a month, a validator can be used to ensure that the value entered is in the range one to twelve. Validators can be combined, to further narrow criteria input data must fill.
JSF provides three different choices of validators:

- UI component validator method.
  Validates the value of a single component. Implemented by overriding the *UIInput.validateValue* method. Not recommended.
- Validator methods in backing beans.
  Used for validating one or more fields of a single form, when the validation code does not need to be reused for other forms.
- Validator classes.
  Associated to the UI components that needs validation, may be combined. Used for validators that are expected to be used more frequently.

Validator Classes

Validator classes must implement the interface *javax.faces.validator.Validator* interface and can be added to UI components that implements the *javax.faces.component.EditableValueHolder* interface. The *javax.faces.validator* package contains four validator classes supplied as part of JSF:

Validator Class	Validator Tag	Validator Id	Comments
DoubleRangeValidator	f:validateDoubleRange	javax.faces.DoubleRange	Checks the value of the corresponding component against specified minimum and/or maximum values of the double type.
LengthValidator	f:validateLength	javax.faces.Length	Checks the number of characters in the String representation of the value of the associated component.
LongRangeValidator	f:validateLongRange	javax.faces.LongRange	Checks the value of the corresponding component against specified minimum and/or maximum values of the long type.
MethodExpressionValidator	f:validator	N/A	Performs validation by executing the *validate* method on an object identified by a MethodExpression.

Note that with the range validators as well as the length validator, one can provide a minimum, a maximum or both a minimum and a maximum.

A validator class is used by either using its corresponding tag or by using the more general <f:validator> tag. The example below shows how the length validator is used in a JSP page.

```
<h:inputText id="name" value="#{person.name}">
    <f:validateLength minimum="5"/>
</h:inputText>
```

Example use of a length validator in a JSP page.

The next example shows how to accomplish the same thing as in the previous example, but using the <f:validator> tag instead. The *validatorId* attribute takes an id of the validator to use, which can be one of the JSF validators or a custom validator.
It also gives an example how to customize the messages displayed if the file is left empty (requiredMessage) and if validation fails, that is, if the length of the name is too short (validatorMessage).

```
<h:inputText id="name" value="#{person.name}" required="true"
    requiredMessage="#{msgs.nameRequired}"
    validatorMessage="#{msgs.nameTooShort}">
    <f:validator validatorId="javax.faces.Length">
        <f:attribute name="minimum" value="5"/>
    </f:validator>
</h:inputText>
```

Example of using the <f:validator> tag to validate length of a text field and how to customize validator messages in a JSP page.

Validator Methods

Methods in plain Java classes can serve as validators for one or more components. This is accomplished by using the *validator* attribute in one of the following tags:

- h:inputHidden
- h:inputSecret
- h:inputText
- h:inputTextArea
- h:selectBooleanCheckbox
- h:selectManyCheckbox
- h:selectManyListbox
- h:selectManyMenu
- h:selectOneListbox
- h:selectOneMenu
- h:selectOneRadio

As an example, the following use of the *validator* attribute will cause the method *validateUser* in the object *user* to be called, in order to validate user input.

```
<h:inputText value="#{user.name}" required="true" validator="#{user.validateUser}"/>
```

Using a custom validation method with the *inputText* tag.

A method with an arbitrary name in a managed bean will serve as validator for the UI component in question. Apart from the name, the method must have the same signature as the *validate* method of *javax.faces.validator.Validator*.
Example of a validator method implementation is show in the listing below.

```
public void validate(FacesContext inContext, UIComponent inComponent, Object inValue)
      throws ValidatorException
{
    FacesMessage theMessage;
    try
    {
        Number theNumber = (Number)inValue;
        if (theNumber.doubleValue() > 0)
        {
            return;
        }

        /* Number did not pass validation. */
        theMessage = new FacesMessage();
        theMessage.setSeverity(FacesMessage.SEVERITY_ERROR);
        theMessage.setSummary(inComponent.getId());
        theMessage.setDetail"Number is not greater than zero!");
    } catch (ClassCastException theException)
    {
        /* Component value was not a Number. */
        theMessage = new FacesMessage();
        theMessage.setSeverity(FacesMessage.SEVERITY_ERROR);
        theMessage.setSummary(inComponent.getId());
        theMessage.setDetail"Please enter a number greater than zero!");
    }
    throw new ValidatorException(theMessage);
}
```

Validator method that ensures that an entered number is greater than zero.

Note that by throwing a *ValidatorException*, the validating code notifies the caller that the value did not pass the validation. The message, an instance of *FacesMessage*, passed to the constructor of *ValidatorException* has three values set; a summary message text, a detail message text and a severity rating. See the next section for how these values can affect the presentation of the message.

Messages

Reference: JSF Specification 6.2

As seen in the section on validators above, validation may fail. In this section it is shown how to notify the user of such failures.

Given the following fragment of a JSF web page, with an input field, an associated validator and a <h:message> tag.

```
<td>
    <h:inputText id="num1_field" value="#{controller.firstNumber}" required="true">
        <f:validateLongRange minimum="0" maximum="999999"/>
    </h:inputText>
/td>
<td>
    <h:message for="num1_field"/>
</td>
```

An input field which require a number in the range 0 to 999,999 and its associated message component.

The following message is displayed when the number is not within the specified range:

```
j_id_jsp_1232077829_1:num1_field: Validation Error:
Specified attribute is not between the expected values of 0 and 999,999.
```

...and this message, if the number is not supplied at all:

```
j_id_jsp_1232077829_1:num1_field: Validation Error: Value is required.
```

An important thing to note about the above example is that if the *required* attribute was not true and the user left the input field empty, the validator would not indicate a validation error! Instead, an exception would be thrown when the model tried to perform the calculation.

Messages related to a certain component can be customized using the *requiredMessage* and *validatorMessage* attributes of the component in question.

```
<td>
    <h:inputText id="num1_field"
                 value="#{controller.firstNumber}"
                 required="true"
                 requiredMessage="You have to enter a number!"
                 validatorMessage="Number must be in the range 0 to 999,999">
        <f:validateLongRange minimum="0" maximum="999999"/>
    </h:inputText>
</td>
<td>
    <h:message for="num1_field"/>
</td>
```

The input field with custom messages added.

Examining the class *javax.faces.application.FacesMessage* one can see that there are two parts in a message; a summary and a detail part. By default, the message tag shows the detail message and hides the summary. This behaviour can be customized using the *showSummary* and *showDetail* attributes.

```
<h:message for="num1_field" showSummary="true" showDetail="false"/>
```

A message tag configured to show only the summary part of messages.

Note that, as stated in the JSF specification document, when customizing messages using the *requiredMessage* or the *validatorMessage* attributes, both the summary message and the detail message in the *FacesMessage* instance that is created when there is a message will be the same.

Additionally, a message also has a severity-level, which can affect how the message is presented. The table below lists the different severity levels, corresponding constants to use when setting the severity of a *FacesMessage* and the attributes of the message tag that can be used to customize the presentation of messages of that severity level.

Severity Level	Constant	Message Tag Attributes
Informational	FacesMessage.Severity.SEVERITY_INFO	infoClass infoStyle
Warning	FacesMessage.Severity.SEVERITY_WARN	warnClass warnStyle
Error	FacesMessage.Severity.SEVERITY_ERROR	errorClass errorStyle
Fatal	FacesMessage.Severity.SEVERITY_FATAL	fatalClass fatalStyle

The class and style message attributes takes strings containing the names of a CSS style class and a CSS style respectively.

Messages are not only created by validators, they can be created by any component in a JSF page. This is the how a message can be created and queued programmatically:

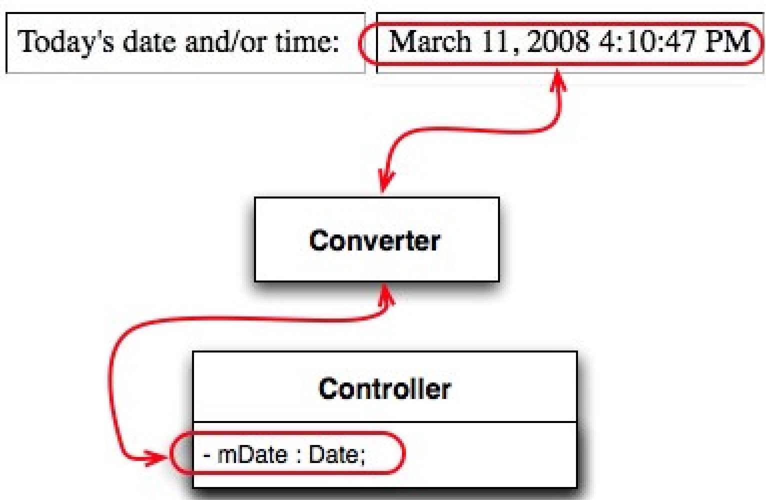
```
FacesMessage theMessage = new FacesMessage();
theMessage.setSummary("Summary validator info message");
theMessage.setDetail("Detail validator info message");
theMessage.setSeverity(FacesMessage.SEVERITY_INFO);
FacesContext.getCurrentInstance().addMessage(null, theMessage);
```

Creating and queueing a JSF message programmatically.

Finally, it should be mentioned that there is a tag, <h:messages>, that can display either all messages or only messages that are not associated with a specific component, depending on the whether the *globalOnly* attribute is false respective true. The latter kind of messages may originate from, for instance, event listeners.

Converters

Reference: JSF Specification 3.3, 4.1.6.2, 9.4.3-9.4.5
Converters are components that can be associated with input or output components in order to convert some data, for instance a date, from a format the computer understands to a human-readable format and vice versa.



A converter converts between the user-readable string and a Date object.

In the picture above, a date is displayed as a string on a web page, but stored in a *Date* object in the controller. A converter is used to generate a text string, showing the date, using the *Date* object. If the text field holding the date had been an input field, the converter would also attempt to convert any text entered in this field by the user to a *Date* object.
A converter may also be configurable, for instance to be able to generate different representations of a date depending on locale, desired format etc.

Attaching Converters

There are three different ways to attach a converter to an UI component.

Using a Specific Converter Tag

A converter can be attached to an UI component by using the appropriate converter tag.

```
<h:outputText value="#{holiday_start.date}">
    <f:convertDateTime/>
</h:outputText>
```

Using the Conveter Attribute

The second way to accomplish something identical to the above is to use the *converter* attribute in the *outputText* tag:

```
<h:outputText value="#{holiday_start.date}" converter="javax.faces.DateTime"/>
```

Note that there is no *javax.faces.DateTime* class in JSP, instead this is an identifier for the *javax.faces.convert.DateTimeConverter* class, see the converter class table below for additional identifiers. It is also possible to configure custom identifiers in the faces-config.xml file.

If the value of the converter attribute is specified using a value expression, like in the example below, then the managed bean must have a property of the type Converter with the name indicated that has already been instantiated.

```
<h:outputText value="#{holiday_start.date}" converter="#{mybean.converterProperty}"/>
```

Using the General Converter Tag

The third way to attach a converter to a component is by using the <f:converter> tag and a converter id:

```
 <h:outputText value="#{holiday_start.date}">
    <f:converter converterId="javax.faces.DateTime"/>
</h:outputText>
```

Note that a converter id is not a fully qualified name of the converter class!
See Converter Classes below for a list of standard converter ids.

Converter Tags

JSF defines the following converter tags, attributes in brackets [] are optional.

Converter Tag	Attributes	Comments
f:convertDateTime	[dateStyle=”{default|short|medium|long|full}”] [locale=”{locale” | string}] [pattern=”pattern”] [timeStyle=”{default|short|medium|long|full}”] [timeZone=”{timeZone| string}”] [type=”{date|time|both}”] [binding=”Value Expression”]	
f:convertNumber	[currencyCode=”currencyCode”] [currencySymbol=”currencySymbol”] [groupingUsed=”{true|false}”] [integerOnly=”{true|false}”] [locale=”locale”] [maxFractionDigits=”maxFractionDigits”] [maxIntegerDigits=”maxIntegerDigits”] [minFractionDigits=”minFractionDigits”] [minIntegerDigits=”minIntegerDigits”] [pattern=”pattern”] [type=”{number|currency|percent}”] [binding=”Value Expression”]	
f:converter	converterId=”converterId” binding=”Value Expression”	The converter must be bound to an object that implements the *javax.faces.convert.Converter* interface.

Converter Classes

Using the converter tag, the following converter classes supplied with the JSF API can be used:

Converter Class	Applies to Type	Converter Id	Comments
BigDecimalConverter	java.math.BigDecimal	javax.faces.BigDecimal	
BigIntegerConverter	java.math.BigInteger	javax.faces.BigInteger	
BooleanConverter	java.lang.Boolean boolean primitive type	javax.faces.Boolean	
ByteConverter	java.lang.Byte byte primitive type	javax.faces.Byte	
CharacterConverter	java.lang.Character char primitive type	javax.faces.Character	
DateTimeConverter	java.util.Date	javax.faces.DateTime	
DoubleConverter	java.lang.Double double primitive type	javax.faces.Double	
EnumConverter	java.lang.Enum enum primitive type	N/A	
FloatConverter	java.lang.Float float primitive type	javax.faces.Float	
IntegerConverter	java.lang.Integer int primitive type	javax.faces.Integer	
LongConverter	java.lang.Long long primitive type	javax.faces.Long	
NumberConverter	java.lang.Number	javax.faces.Number	Has more sophisticated conversion options, see API!
ShortConverter	java.lang.Short short primitive type	javax.faces.Short	

All of the above conversion classes are located in the *javax.faces.convert* package.

Converter Messages

If a converter fails to convert data to/from string representation, it will issue a message. Converter messages behave in a similar manner to validator messages:

- Messages from a single UI component can be displayed using the <h:message> tag.
- The message from a UI component can be customized using the *converterMessage* attribute.

```
<td>
    <h:inputText id="dateField" value="#{dateController.date}">
        <f:convertDateTime dateStyle="short" type="date"/>
    </h:inputText>
    <h:message for="dateField"/>
</td>
```

Input field that accepts input that is stored as a *Date* object in a managed bean.

In the above configuration, the following message will be displayed if the user enters a string that the converter is not able to interpret into a date:

```
_id_jsp_1340549415_2:dateField: '14/14/08' could not be understood as a date. Example:
3/14/08
```

An important thing to note about the above example is that if the *required* attribute is not true, or left out, and the user left the input field empty, the converter will not indicate a conversion failure. Instead, an exception might be thrown later, when the user input is processed.

Messages related to a certain component can be customized using the *requiredMessage* and converterMessage attributes of the component in question.

```
<td>
    <h:inputText id="dateField" value="#{dateController.date}" required="true"
                 converterMessage="Failed to recognize input as a date"
                 requiredMessage="A date must be supplied!">
        <f:convertDateTime dateStyle="short" type="date"/>
    </h:inputText>
    <h:message for="dateField"/>
</td>
```

The input field with custom messages and the required attribute added.

Please refer to the last part of the section Validator Messages on how the presentation of messages can be configured using CSS (cascading style sheets) etc.

Renderers

Reference: JSF Specification 8

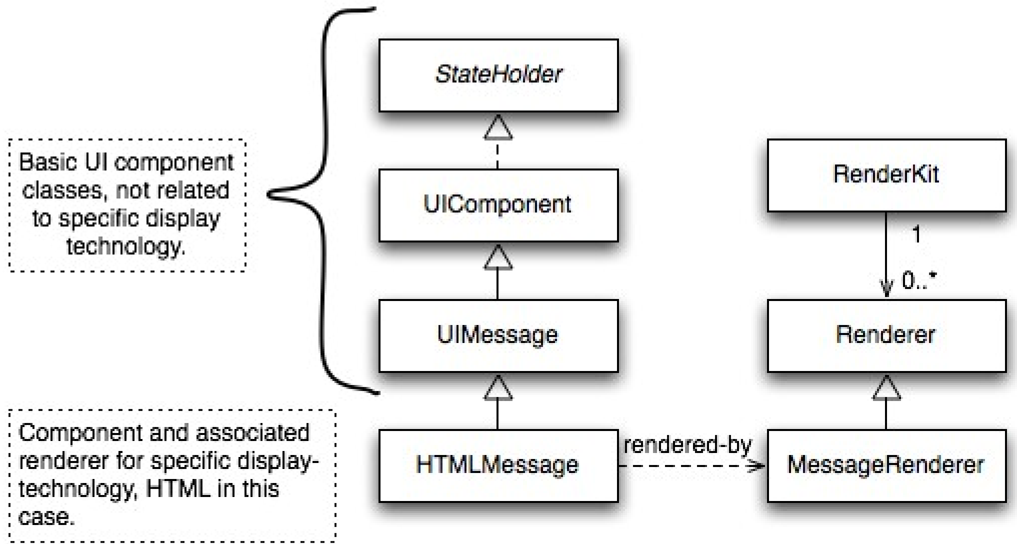
JSF supports the following two models for deciding component values from incoming requests and encoding values into outgoing responses:

- Direct Implementation Model
  Each component is responsible for decoding and encoding itself.
- Delegated Implementation Model
  Decoding and encoding of components is delegated to appropriate renderer.

Decoding of a component consists of, in the case of HTML, extracting any (new) values for the component enclosed in the request in question.

> **Background:**
> JSF uses view state saving, either on the server or client side. This means that the state of the user interface is serialized, encoded and either enclosed with the response or stored on the server. Later, when the user submits a form, the server uses the view state to reconstruct the view. See the section Request Processing Life Cycle for more information!

Below is an example of the relationship between UI components and renderers shown for the message component. Some classes have been left out of the class hierarchy, in order to focus on the principles.



Example of relationship between the message UI component and its renderer.

Events and Listeners

There are three standard event types in JSF:

- Action Events
- Value Change Events
- Phase Events

Action Events

Reference: JSF Specification 3.4.2, 3.4.3, 5.2, 7.1.1, 7.3, 9.2.6, 9.4.1, 9.4.12
An action event represents an activation of a UI component, for instance, the user presses a button on a JSF web page.
Action events in JSF are represented by instances of the *javax.faces.event.ActionEvent* class.
The following methods are available to register an action listener for a component:
There are two ways to register an action listener for a component:

- Using an *actionListener* attribute.
- Using the <f:actionListener> tag.

The actionListener *Attribute*

The following code shows how to register an action listener in a managed bean on a command button using the *actionListener* attribute.

```
<h:commandButton value="Commit Request" actionListener="#{bean.doCommit}"/>
```

Registering an action listener using the *actionListener* attribute.

The method *doCommit(ActionEvent)* of the managed bean will be called when the user presses the button. A potential drawback is that only a single action listener may be registered on the component. The skeleton of the *doCommit* method looks like this:

```
public class ManagedBean
{
    ...
    public void doCommit(ActionEvent inAction)
    {
        // Handle the button-click here.
    }
    ...
}
```

Skeleton of the *doCommit* method handling clicks on the “Commit Request” button.

Note that the class in which the *doCommit* method is located does not have to implement any particular interface.

The <f:actionListener> Tag

The code below accomplishes the same thing as the example above, that is, registering an action listener on a command button. This time, the <f:actionListener> tag is used instead.

```
<h:commandButton value="Commit Request">
    <f:actionListener type="com.ivan.ManagedBean"/>
</h:commandButton>
```

Registering an action listener using the <f:actionListener> tag.

When using the <f:actionListener> tag, multiple action listeners can be registered for a single component.
The *processAction(ActionEvent)* method in an object of the class *com.ivan.ManagedBean* will be called when the user clicks the button. The *type* attribute specifies a Java class that must implement the *javax.faces.event.ActionListener* interface. Alternatively, the *binding* attribute may be used to specify a managed bean or an attribute that refers to an object that implements the *javax.faces.event.ActionListener* interface.

```
public class ManagedBean implements ActionListener
{
    ...
    public void processAction(ActionEvent inEvent) throws AbortProcessingException
    {
        // Handle events here.
    }
    ...
}
```

Skeleton of the *processAction* method and the class in which it is located.

The table below specifies the behaviour of the <f:actionListener> tag when one, or both, of the attributes are supplied.

Attribute(s) Supplied	Description
type	Create a new instance of the type specified each time the action listener is to be invoked and invoke its *processAction* method.
binding	If value expression evaluates to null, generate a faces message. If value expression evaluates to an object that implements *javax.faces.event.ActionListener*, invoke its *processAction* methods.
type, binding	If the binding value expression evaluates to null, create a new instance of the type specified and use it as action listener, otherwise use the existing object. Invoke the *processAction* method in the action listener.

The <f:setPropertyActionListener> Tag

<f:setPropertyActionListener> is a specialized action listener used to set a property. An example of its use would be to copy the value in a scoped variable to the property of a managed bean when a web page is submitted.

```
<h:commandButton value="Magic Button">
    <f:actionListener binding="#{mybean}"/>
    <f:actionListener type="com.ivan.MyActionListener"/>
    <f:setPropertyActionListener value="#{mybean.inputString}"
      target="#{otherbean.inputString}"/>
</h:commandButton>
```

A command button with two regular action listeners and a set property action listener.

In the above code-fragment, the command button has a set property action listener registered to it. When the button is clicked, this listener will cause the value from the property *inputString* of the managed bean *mybean* to be copied to the property *inputString* of the managed bean *otherbean*. Any number of <f:setPropertyActionListener> tags can be used with a single component.

Invocation Order

As described in the *javax.faces.component.UICommand* class, listeners will be invoked in the following order:

- Action listeners registered using the *actionListener* attribute.
  For instance, in the <h:commandButton> tag, as shown above.
- Action listeners registered using the <f:actionListener> tag.
  These are invoked in the order declared/registered.
- The application action listener.
- Application action methods.

Note that if a custom application action listener is used and the default application action listener is not invoked, then application actions will not be invoked.

An application listener that is used as a custom application action listener may be implemented like this, to ensure invocation of the default application action listener:

```
public class AppActionListener implements ActionListener
{
    private ActionListener mParent;

    public AppActionListener(ActionListener inParent)
    {
        mParent = inParent;
    }

    public void processAction(ActionEvent inEvent) throws AbortProcessingException
    {
        // Process action event here.
        mParent.processAction(inEvent);
    }
}
```

Class implementing a custom application action listener that also invokes
the parent, i.e. default, application action listener.

Application Actions

There is another type of action listener, called "application action", that has the ability to affect navigation between pages in a JSF application. An application action has the following characteristics:

- Returns an *Object* that indicates the outcome of the operation.
- Takes no parameters.
- Does not require the class to implement any particular interface.
- Must be public.

The following code registers an application action on a command button.

```
<h:commandButton value="OK" type="submit" action="#{controller.doOK}"/>
```

A command button using an action listener that affects navigation.

When the button is clicked, the *doOK* application action method in the managed bean will be called. The skeleton of the *doOK* method will look like this:

```
public String doOK()
{
   // Handle button clicks here, perhaps with different outcome.
   return "success";
}
```

Application action method *doOK* in the managed bean.

The method above returns the outcome "success". This outcome may affect navigation, as we will see in the Navigation section later.
If all an application action method does is return the outcome "success", it can be eliminated and the command button can be written like this:

```
<h:commandButton value="OK" type="submit" action="succcess"/>
```

A command button with a static outcome.

The value *null* may also be returned by an application action method, in which case the view will redraw itself. This can be useful if, for instance, some setting that affects the way the view is drawn has been changed by the action method.
Action listeners and application actions can configured at the same time for a component.

```
<h:commandButton value="OK" type="submit" action="#{bean.doOK}"
actionListener="#{bean.myActionListener}"/>
```

A command button with both an action listener and an application action.

This technique can be useful when it is needed to, for instance, determine in which section of a picture the user clicked before letting this affect the navigation.

Value Change Events

Reference: JSF Specification 2.2.2, 3.2.6, 3.4.2, 3.4.3, 5.2, 9.2.6, 9.4.18
Value change events occur when the value of a UI component changes from one value to another, valid, value.
Value change events in JSF are represented by instances of the *javax.faces.event.ValueChangeEvent* class.
As with action events, there are two ways to register a value change event listener for a component:

- Using the *valueChangeListener* attribute.
- Using the <f:valueChangeListener> tag.

The valueChangeListener Attribute

The valueChangeListener attribute allows for registration one single value change listener on a component.

```
<h:inputText value="#{bean.dataString}"
    valueChangeListener="#{bean.stringValueChange}"/>
```

Registering a value change listener using the *valueChangeListener* attribute.

In this case, the method stringValueChange*(ValueChangeEvent)* of the managed bean will be called when the user has entered a new value that passes validation. The managed bean is not required to implement any particular interface.

The <f:valueChangeListener> Tag

The <f:valueChangeListener> tag does not have the limit of only being able to register one single value change listener on a component.

```
<h:inputText value="#{bean.dataString}">
    <f:valueChangeListener type="com.ivan.ManagedBean"/>
</h:inputText>
```

Registering a value change listener using the <f:valueChangeListener> tag.

As with action listener, the type attribute specifies a Java class. This time, the class specified by the *type* attribute has to implement the *javax.faces.event.ValueChangeListener* interface. When the user has entered a new value that passes validation, the *processValueChange(ValueChangeEvent)* method will be called on an instance of the class.

Phase Events

Reference: JSF Specification 4.1.17.4, 9.4.9, 11.2, 11.3
Phase events are events that are generated before and after each of the phases in the Request Processing Life Cycle of JSF. In JSF, a phase event is represented by an instance of the *javax.faces.event.PhaseEvent*.

Phase Listener

In order to be able to listen for phase events, a phase listener is needed. A phase listener is a class that implements the *javax.faces.event.PhaseListener* interface. The interface contains the following methods:

Method	Description
void afterPhase(PhaseEvent)	Receives notifications that a phase has been completed.
void beforePhase(PhaseEvent)	Receives notifications that a phase is about to begin.
PhaseId getPhaseId()	Returns the id of the phase the listener wants to receive notifications about.

See the implementation of the managed bean class in appendix 1 for an example how a phase listener is implemented.

Registering a Phase Listener

A phase listener can be registered in two ways; either programmatically in a managed bean or by using the <f:phaseListener> JSF core tag.
FacesContext theContext = FacesContext.getCurrentInstance();

```
...
FacesContext.getCurrentInstance().getViewRoot().addPhaseListener(this);
...
```

Registering a phase listener programmatically from the class that implements the *javax.faces.event.PhaseListener* interface.

```
    ...
    <f:view>
        <f:phaseListener binding="#{bean}"/>
        ...
    </f:view>
    ....
```

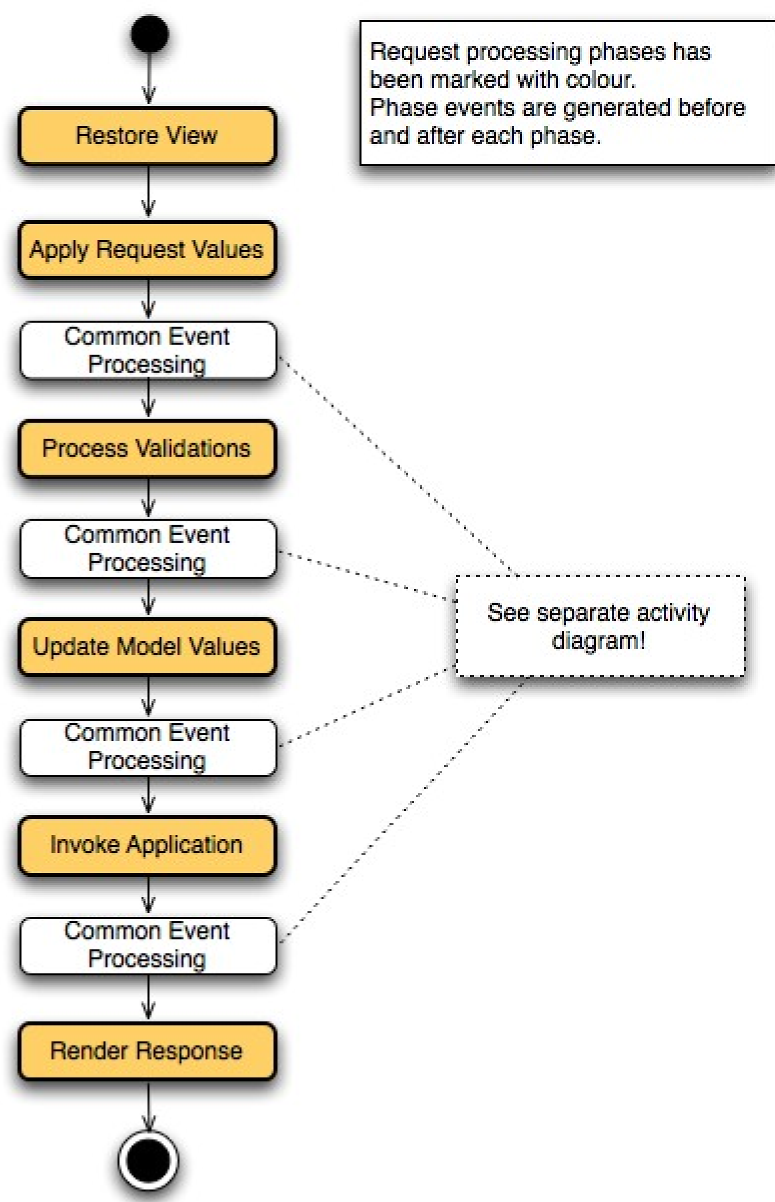
Registering a phase listener using the <f:phaseListener> tag in a JSF web page.
Note that a managed bean implementing the *javax.faces.event.PhaseListener* interface being bound to the name bean is required.

Request Processing Life Cycle

Reference: JSF Specification 2.2, 2.3
Every request that involves a JSF component tree is processed using a well-defined request processing life cycle that consists of a number of phases. The standard request processing life cycle consists of the following phases:

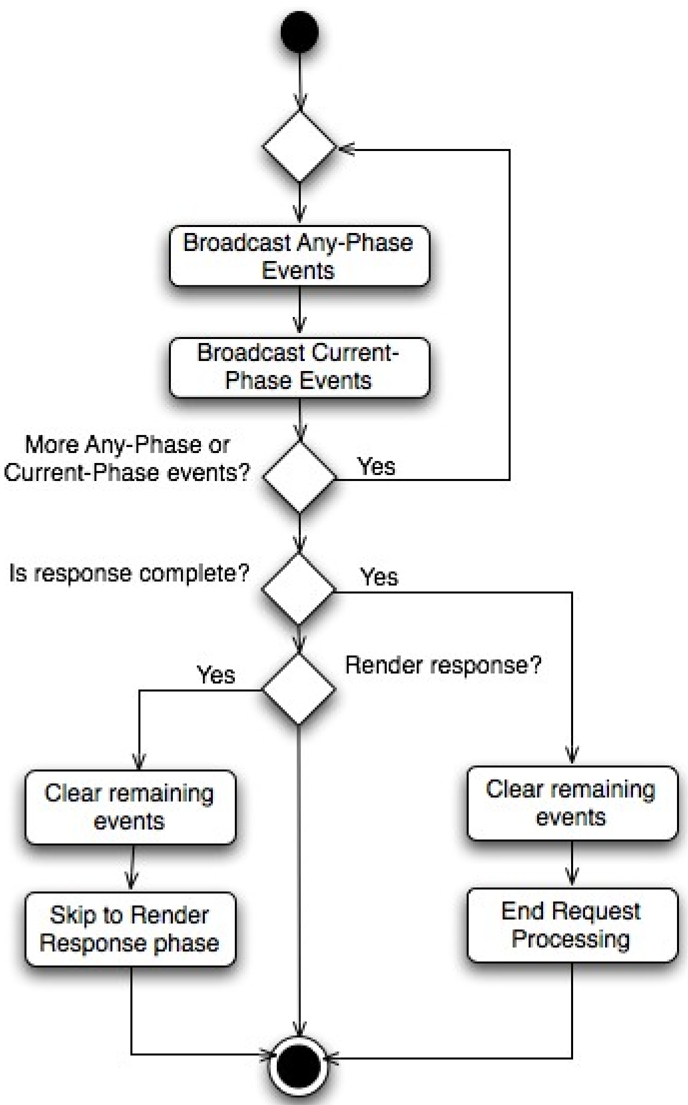
- Restore View
- Apply Request Values
- Process Validations
- Update Model Values
- Invoke Application
- Render Response

An activity diagram showing an overview of the request processing life cycle is shown below.



Overview of the JSF Request Processing Life Cycle.

The common event processing taking place after every phase, except for the Restore View and the Render Response phases, is outlined in the following activity diagram.

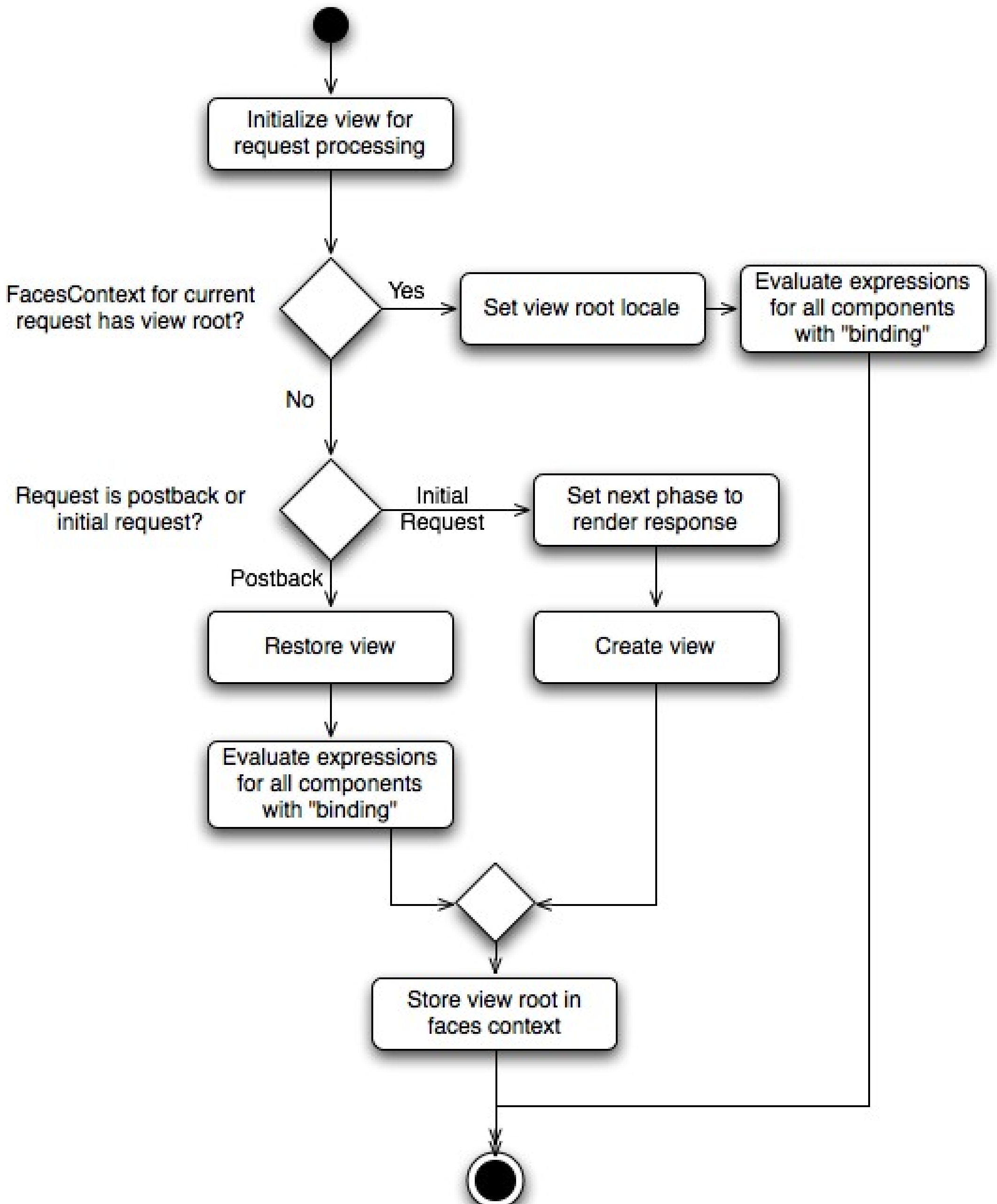


Common event processing between phases in the JSP Request Processing Life Cycle.

The check for more events after events have been broadcasted is performed in case the broadcasting of events give rise to more events. In the current JSF implementation from Sun, this check is only performed once.
The response being "complete" means that it has already been rendered, or that rendering is not necessary, for instance if a redirect occurs. A response is marked as being complete by calling *responseComplete()* on the current *FacesContext* instance.
The response is marked for rendering by calling *renderResponse()* on the current *FacesContext* instance. This causes the request processing to go to the Render Response phase, skipping any remaining phases, once the current phase is completed.
For an example program that shows the different phases of a JSF request and associated events, see appendix 1.

Restore View

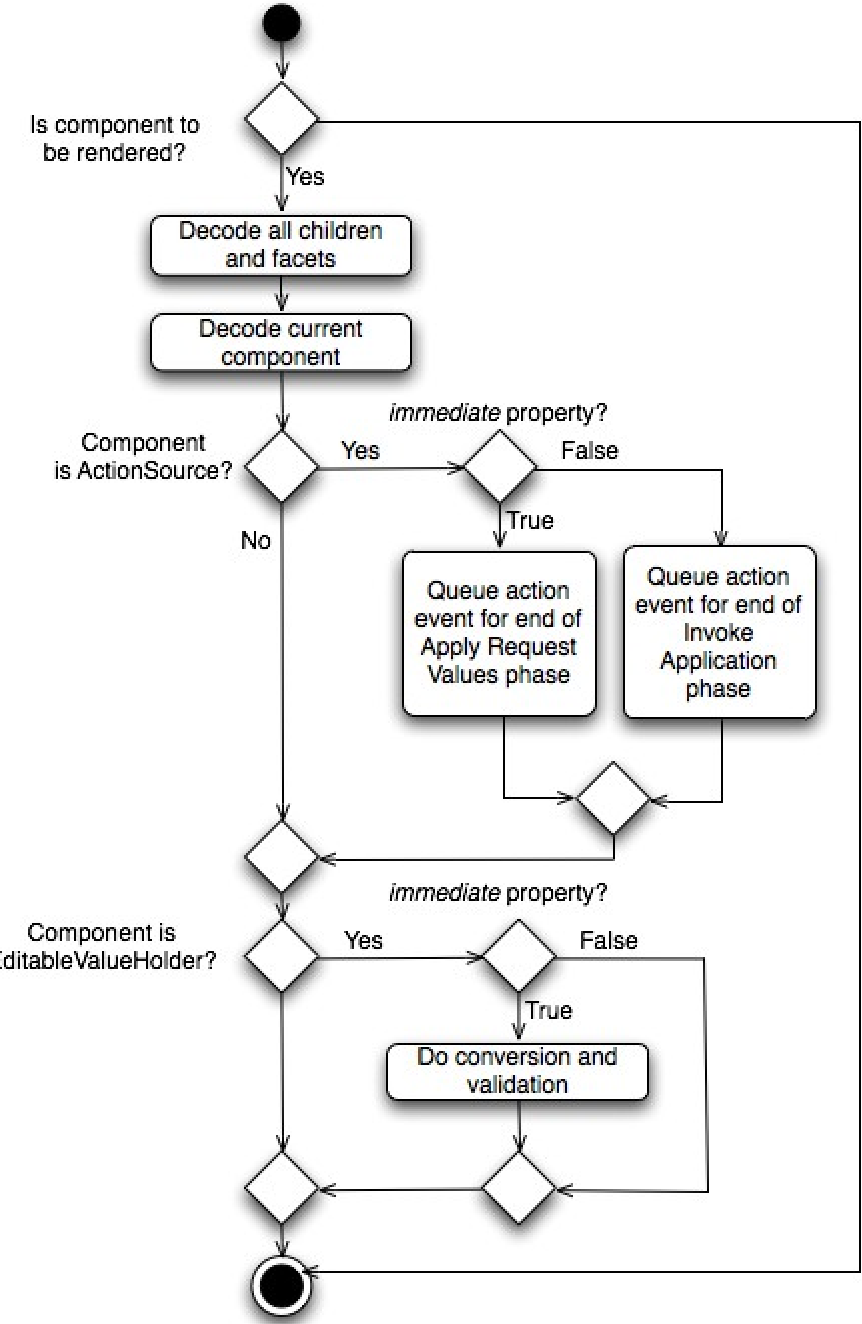
The purpose of the restore view phase is to prepare the view of the page the request concerns, or, if no such view exists, create a new view. Any values of the components in the view are also restored during this phase.



Overview of the Restore View phase in the JSF request processing life cycle.

Apply Request Values

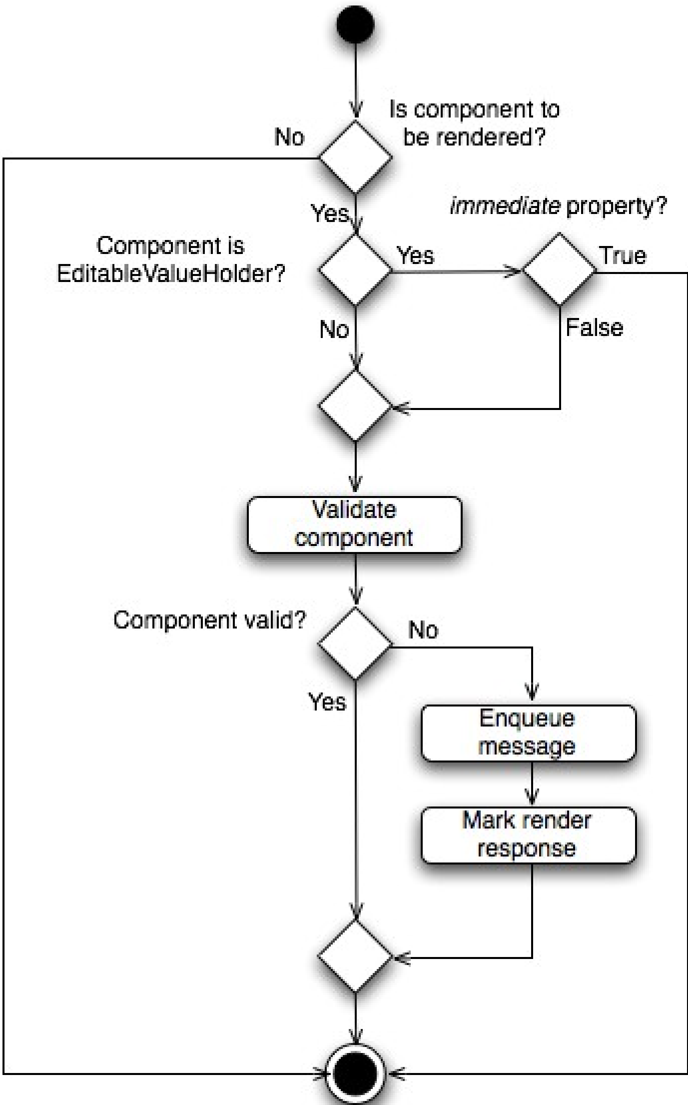
During the Apply Request Values phase, each component of the view is given an opportunity to update its state using the information included with the current request.
The activity diagram below shows the general procedure of a single component, during this phase.



General procedure for a single component during the Apply Request Values phase.

Process Validations

The purpose of the Process Validations phase is to ensure that all the components in the view have valid values. The action diagram below gives an overview of the procedure for a single component.

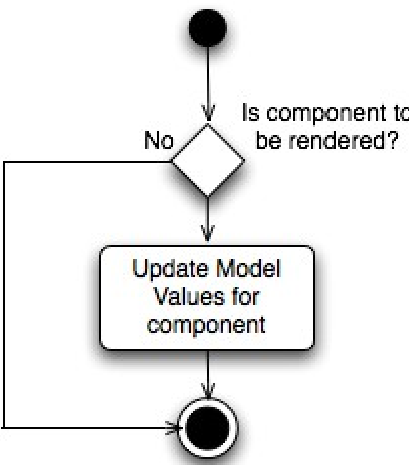


Validation procedure for a single component during the Process Validations phase.

Note that if validation fails for any component during this phase, messages are enqueued and the request processing skips to the Render Response phase, once the Process Validation phase has completed, and error messages will be displayed.

Update Model Values

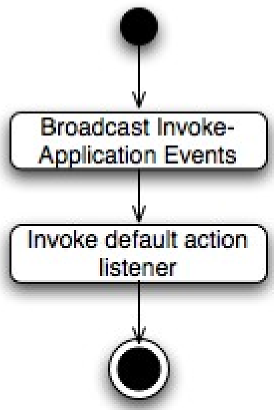
During the Update Model Values phase, properties in model data objects are updated with the current values from respective component. Model data objects can be managed beans etc.
Note that a "model data object" does not necessarily need to be part of the model, as defined in the MVC design pattern. Rather, it is more likely to be a controller.



Procedure for updating model value(s) for a single component during the Update Model Values phase.

Invoke Application

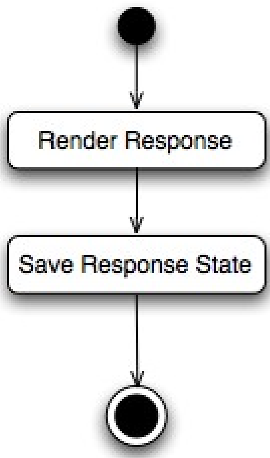
Having ensured that data is valid and then passed it on to the model data objects, the application logic can now be invoked during the Invoke Application phase.



Actions taken during the Invoke Application phase.

Render Response

As the final step of the JSF Request Processing Life Cycle, the response is rendered and sent back to the client.



Procedure of the Render Response phase.

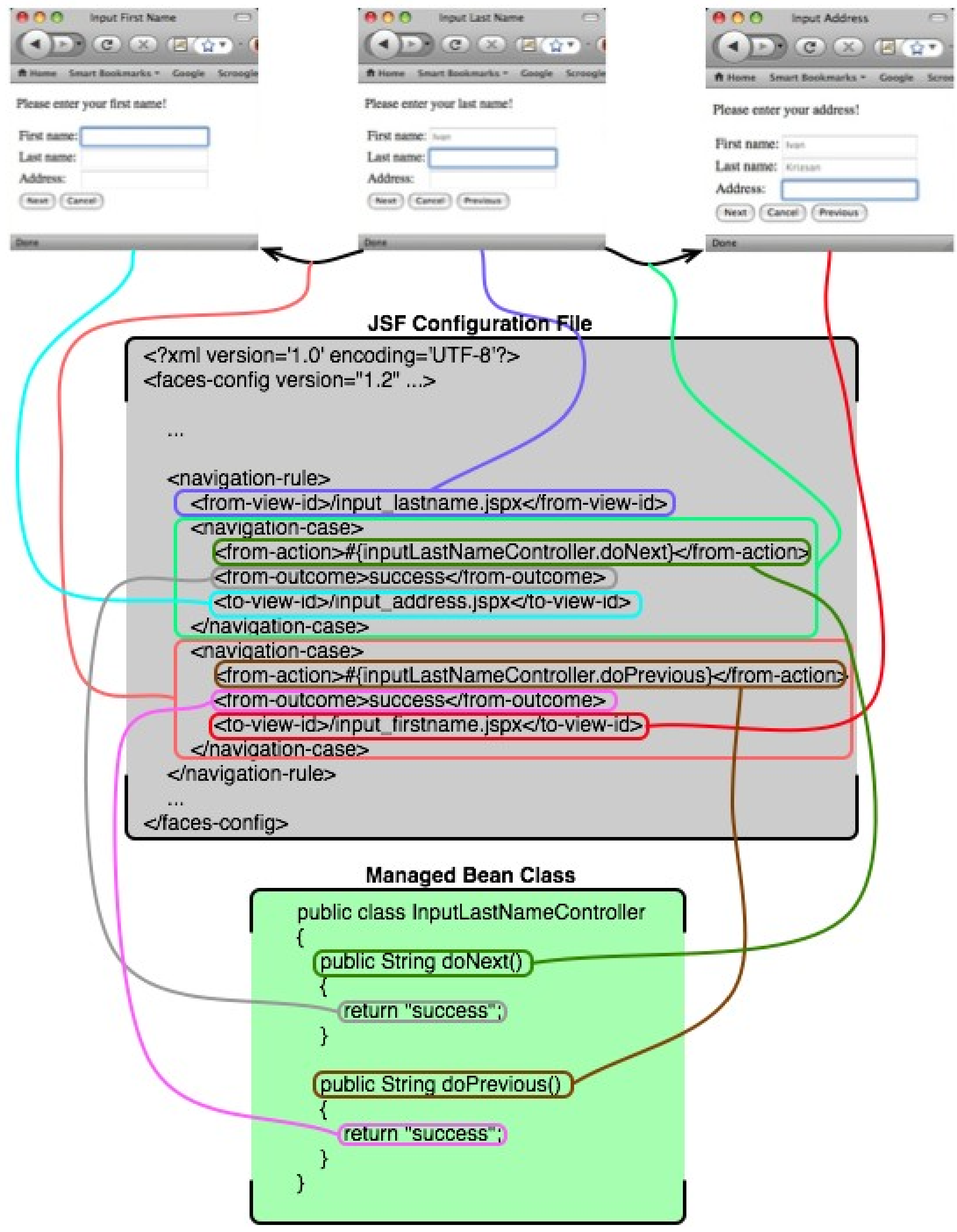
Navigation

Reference: JSF Specification 7.4

Navigation between pages in JSF applications are controlled using a combination of the JSF configuration file and outcomes from action events. Outcomes from action events are strings such as “success” or “cancel” etc. and has been described in the Action Events section.

Some IDEs allow for graphical editing of the navigation rules, but that is beyond the scope of this document.

Lets take a look at what navigation rules may look like and what the different parts of a navigation rule corresponds to.



Navigation configuration for the Input Last Name page in the JSF configuration file and corresponding elements.

Each <navigation-rule> contains one or more navigational alternatives from one, or several, views. Using the <from-view-id> element, there are three different ways of specifying the view, or views, that are the origin of the navigation. Priority according to the order listed.

- <from-view-id>/index.jsp</from-view-id>
  Will only match one single view; in this example the web page “index.jsp”.
- <from-view-id>/test*</from-view-id>
  Will match any view identifier that starts with “test”, for instance “test_page.jsp”.
- <from-view-id>*</from-view-id> or <from-view-id>/*</from-view-id>
  Will match any view identifier, that is, this navigational rule may apply to any view in the JSF application.

The <from-view-id> element may also be left out completely, which will match all views.

Each <navigation-case> then, optionally, specifies an action and a outcome of the action, using the <from-action> and <from-outcome> elements respectively, that causes the navigation to the destination specified by the required <to-view-id> element.
An action is, as in the above figure, a method in a managed bean.
An outcome of an action is an arbitrary string.

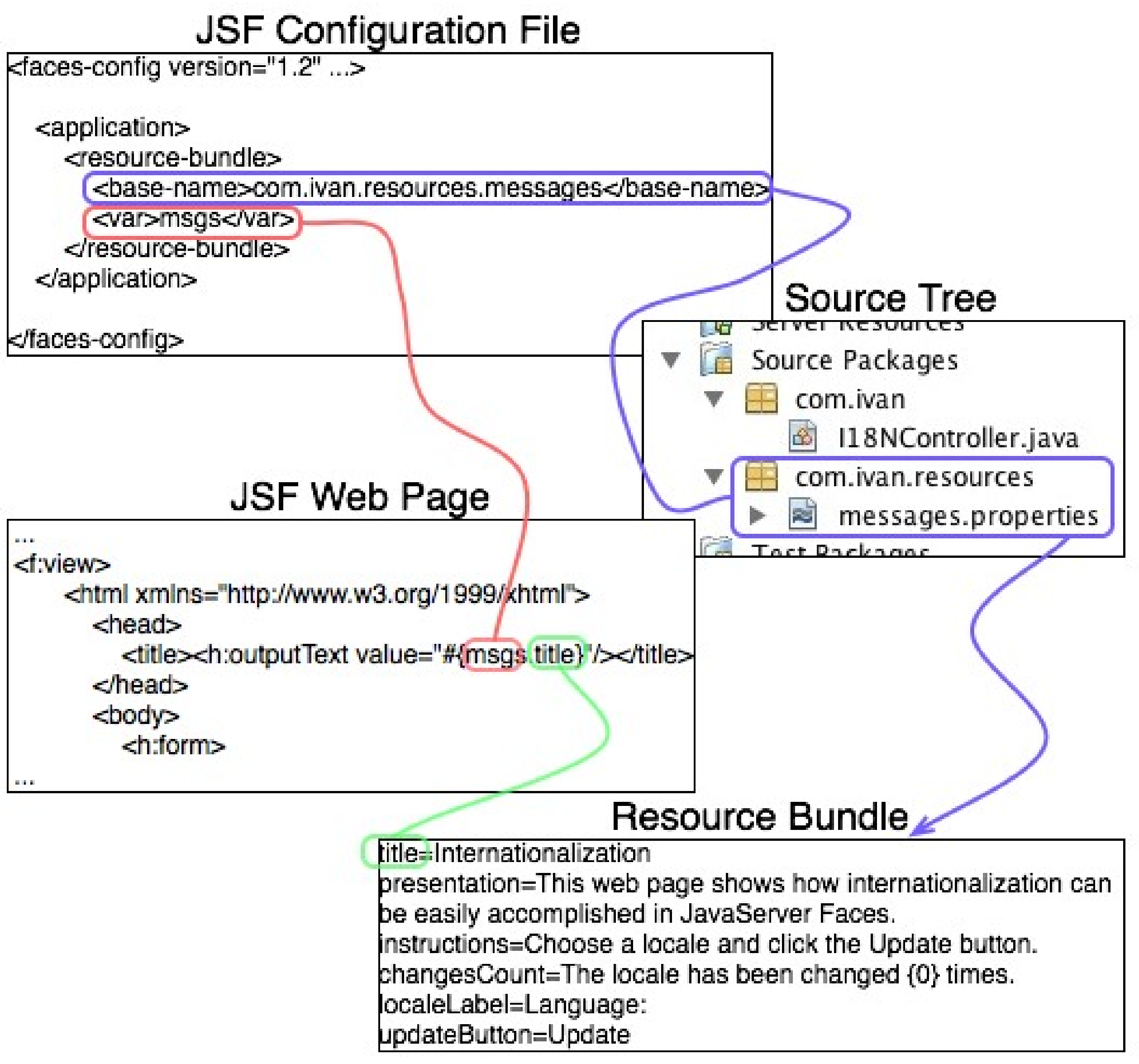
The optional <redirect/> element enables HTTP redirection to the destination view. This means the URL of the destination view will be visible in the browser. If left out, the request will be forwarded to the destination view. Note that when redirecting a HTTP request, any request scoped data is lost, since a new request is made.

For a detailed example on how to do navigation in JSF, please refer to appendix 3!

Localization and Internationalization

Reference: JSP Standard Tag Library Specification v1.2 chapter 8, JSP Specification v2.1 chapter 4, JSF Specification 2.5.2

The JSF support for internationalization is not only useful when an application is to be able to present messages in different languages, but also very helpful to aid in organizing the messages an application is to present.
Finally, a sample program that allows the user to select the current locale, choosing from the locales supported by the JSF application, is available in appendix 4!
The following picture shows the relationships between the JSF configuration file, a resource bundle containing internationalized messages, the location of the resource bundle file in the source tree and the JSF web page in which the bundle is used.



Different elements of JSF internationalization and localization.

Resource Bundles

Message strings are commonly entered into one or more property files in which every message is given a key. Such a file is called a resource bundle and, when used, becomes an instance of the *java.util.ResourceBundle* class (or a subclass). Resource bundles are to be located in a regular Java-package, as will be seen later.
Note that with JSF, keys in resource bundles must not contain JSTL reserved characters, such as “.”.

```
title=Internationalization
instructions=Choose a locale and click the Update button.
changesCount=The locale has been changed {0} times.
localeLabel=Language:
updateButton=Update
```

Sample messages property file; message-keys to the left and messages to the right.

Messages files are named depending on the language used in the messages. The default language message file can have an arbitrary name, for instance “messages”, with the suffix “properties”. If the application also is to support German localization, there will be another file with the name “messages_de.properties” containing the messages of the application in German.
Language codes are defined by the ISO-639 standard. See the *java.util.Locale* class for more information.

Accessing Resource Bundles

JSF needs to be told which resource bundle(s) to use and how the programmer wants to access these. This can be accomplished either by using the JSF configuration file or by using the <f:loadBundle> tag.

```
...
    <application>
        <resource-bundle>
            <base-name>com.ivan.resources.messages</base-name>
            <var>msgs</var>
        </resource-bundle>
    </application>
...
```

Configuring access to a resource bundle in the JSF configuration file.

```
...
    <f:loadBundle basename="com.ivan.resources.messages" var="msgs"/>
...
```

Configuring access to a resource bundle using the <f:loadBundle> tag.

The difference is that when configuring access to a resource bundle using the JSF configuration file, the resource bundle is then accessible from all the JSF web pages of the application. When using the <f:loadBundle> tag, it must be included in every JSF web page that wants to access the resource bundle.

Configuring Supported Locales

It is also possible to, using the JSF configuration file, tell JSF which locales are available and which one is the default locale for the application. Default locale is the locale that is placed in the resource bundle file which name does not have a language code appended, for instance “message.properties” (compared to “messages_de.properties”).
Say, for instance, that our application has messages in English, German and Swedish, and that English is the default language. This would be configured in the JSF configuration file like this:

```
...
    <application>
        ...

        <locale-config>
            <default-locale>en</default-locale>
            <supported-locale>en</supported-locale>
            <supported-locale>de</supported-locale>
            <supported-locale>sv</supported-locale>
        </locale-config>
    </application>
...
```

Configuring the available locales and the default locale in the JSF configuration file.

The corresponding files, “messages.properties”, “messages_de.properties” and “messages_sv.properties”, also have to be present.

Retrieving the Message

If we have configured the resource bundle as described in the above sections, the string "Internationalization" can be retrieved and output by the following code fragment:

```
...
<h:outputText value="#{msgs.title}"/>
...
```

Retrieving a localized message from a resource bundle in JSF.

If the resource bundle has been configured for multiple languages, the output depends on the current locale.

Controlling the Locale

The current locale can be retrieved and set in JSF, allowing for programmatic configuration of the language in which the web pages are displayed. For a complete example program that allows the user to select the current locale, see appendix 4!
First of all, let's take a look at how to retrieve the current locale. There are actually two "current locales" that can be retrieved; the current JSF locale and the browser's current locale.

Browser's Current Locale

The browser's current locale can be retrieved by retrieving the locale from the request object. Note that JSF cannot request the locale in the request, so it has to be copied into a variable.

```
<c:set var="browserLocale" scope="request" value="${pageContext.request.locale}"/>
<h:outputFormat value="#{msgs.localeCodeLabel}">
    <f:param value="#{requestScope.browserLocale}"/>
</h:outputFormat>
```

Retrieving the browser's current locale in a JSP.

The external context in the faces context has a special method to retrieve the request locale:

```
...
Locale theBrowserLocale =
    FacesContext.getCurrentInstance().getExternalContext().getRequestLocale();
...
```

Retrieving the browser's current locale in Java code.

The browser's locale cannot be changed, unless the entire request object is replaced.

JSF's Current Locale

The locale used when rendering the current view is contained in the view root and may be both retrieved and set.

```
...
Locale theCurrentJSFLocale =
FacesContext.getCurrentInstance().getViewRoot().getLocale();
...
FacesContext.getCurrentInstance().getViewRoot().setLocale(theLocale);
...
```

Retrieving and setting JSF's current locale from the view root.

Retrieving Locale Configuration

As seen in the section Configuring Supported Locales above, the locales supported by a JSF application can be configured in the JSF configuration file. Messages are, as described in the Resource Bundles section, placed in separate files. Thus, the languages supported by a JSF application can be changed without requiring any changes in the code.
How to write code that takes this kind of flexibility into account?
The default locale and the supported locales can be retrieved in, for instance, a managed bean using these snippets of code.

```
...
/* Retrieve all supported locales, as configured in the JSF configuration file. */
Iterator<Locale> theLocalesIter = FacesContext.getCurrentInstance().getApplication().
    getSupportedLocales();
...
/* Retrieve the default locale, as configured in the JSF configuration file. */
Locale theDefaultLocale = FacesContext.getCurrentInstance().getApplication().
    getDefaultLocale();
...
```

Retrieving the supported locales and default locale of a JSF application.

Core Tags and User Interface Components in Detail

This section goes into greater detail on some of the user interface components and core tags and shows some examples on how these tags can be used.

Attributes and Custom Attributes

The <f:attribute> tag allows the programmer to add attributes to its parent UI component, like in this example:

```
<h:panelGroup layout="block">
    <f:attribute name="style" value="color : red"/>
    <h:outputText value="Some text in a panel group"/>
</h:panelGroup>
```

Adding a style attribute to the panel group using the <f:attribute> JSF core tag.

The normal way to write the above would be:

```
<h:panelGroup layout="block" style=”color : red”>
    <h:outputText value="Some text in a panel group"/>
</h:panelGroup>
```

Adding a style attribute to the panel group using a more common way.

In this case, little is gained by using the <f:attribute> JSF core tag. However, there are cases where one might want to add non-standard attributes to a tag.
IDEs like NetBeans will complain if you try to add custom attributes to a tag, like in the following example:

```
<h:panelGroup layout="block" myattr="test">
    <h:outputText value="Some text in a panel group"/>
</h:panelGroup>
```

Adding a custom attribute to a tag – this won't work.

Later, when trying to run the web application, the following error will occur:

```
HTTP Status 500 -
type Exception report
message
description The server encountered an internal error () that prevented it from
fulfilling this request.
exception
org.apache.jasper.JasperException: /welcomeJSF.jsp(28,12) Attribute myattr invalid for
tag panelGroup according to TLD

org.apache.jasper.compiler.DefaultErrorHandler.jspError(DefaultErrorHandler.java:40)
       ...
```

Result of trying to add custom attributes to a tag.

This is where the <f:attribute> tag comes in handy. By rewriting the code that produced the error using the <f:attribute> tag, no error will occur.

```
<h:panelGroup layout="block" myattr="test">
    <f:attribute name=”myattr” value=”test”/>
    <h:outputText value="Some text in a panel group"/>
</h:panelGroup>
```

Adding a custom attribute to a tag and avoiding errors.

This attribute can later be retrieve from code in a managed bean like this:

```
public String doIt()
    {
        Map<String, Object> theAttributes;

        theAttributes = mCustAttrComp.getAttributes();
        if (theAttributes.size() != 0)
        {
            System.out.println("The component has the following attributes:");
            for (String theKey : theAttributes.keySet())
            {
                System.out.println("Key: " + theKey + " with the value: " +
                    theAttributes.get(theKey));
            }
        } else
        {
            System.out.println("The component has no attributes.");
        }

        return "success";
    }
```

Method in a managed bean listing all attributes of the UI component bound to *mCustAttrComp*.

Making Selections

JSF has a number of UI components that presents a list of multiple items to the user, allowing him, or her, to select one, or many, of the items in the list. In this section those UI components will be introduced in greater detail and examples on how to use them will be shown.

Single Checkboxes

A single checkbox allows the user to check, or un-check, an alternative. The one thing that sets the single checkbox apart in JSF is that its label has to be provided separately. In order to toggle the checkbox when the label is clicked, the label also has to be linked to the checkbox in question.

```
<h:selectBooleanCheckbox id="checkbox1" value="#{mymanagedbean.booleanflag}"/>
<h:outputLabel for="checkbox1">
    <h:outputText value="Checkbox label text"/>
</h:outputLabel>
```

Creating a checkbox with a label. Clicking the label will also toggle the checkbox.

Selecting from a List

The following UI components allow for selecting one or more items from a list of items.

- h:selectOneListbox
- h:selectOneMenu
- h:selectOneRadio
- h:selectManyCheckbox
- h:selectManyListbox
- h:selectManyMenu

The JSF tags that start with “selectOne” selects one single item, while the tags starting with “selectMany” allows for selection of multiple items. This affects the variable used to store the selection(s) in in the managed bean supporting the web page. When selecting a single item, the value expression provided using the *value* attribute refers to a variable of the type *String* or *int* etc. in a managed bean. When selecting multiple items, the value expression provided using the *value* attribute must instead refer to a variable being an array of *String* or *int* etc.

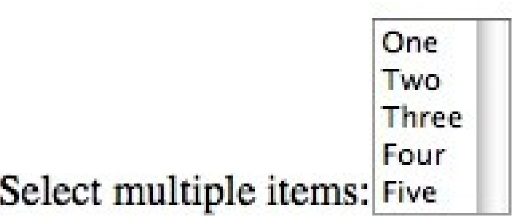
The list of items to select from is created by adding one or more <f:selectItem> and/or <f:selectItems> tags as children to the select tag. These tags can have hardcoded values or retrieve data from a managed bean.

Hardcoded List Items

The following code produce a list box according to the picture below. Note that the appearance can vary from browser to browser!

```
<h:outputText value="Select multiple items:"/>
<h:selectManyListbox id="sel_menu3" value="#{mybean.menuThreeSelection}">
    <f:selectItem itemLabel="One" itemValue="1"/>
    <f:selectItem itemLabel="Two" itemValue="2"/>
    <f:selectItem itemLabel="Three" itemValue="3"/>
    <f:selectItem itemLabel="Four" itemValue="4"/>
    <f:selectItem itemLabel="Five" itemValue="5"/>
</h:selectManyListbox>
```

List box allowing for selecting of multiple items from a list of hardcoded values.



The above list box appearing in a browser.

In the managed bean, the property *menuThreeSelection* can be of the type *String[]* or *int[]* and, depending on the selection(s) made by the user, will contain an array containing items with values “1” (or 1) to “5” (or 5). JSF will automatically take care of the conversion to the correct type.

If the conversion between the *itemValue* and the variable type in the managed bean fails, there will be a faces message saying so. If there is no UI component displaying messages on the web page, the only place where the message can be seen is the application server log. To ensure discovery of problems like this, place a temporary <h:messages> component somewhere on the web page when developing or keep an eye on the log.

Fallback List Items

The next example shows a list allowing the selection of one item. The first item in the list will try to retrieve data from an attribute in the supporting managed bean. However, if this attribute does not exist or is null, the supplied, hardcoded, values will be used.

```
<h:selectOneListbox id="sel_menu2" value="#{mybean.menuTwoSelection}">
    <f:selectItem value="#{mybean.menuTwoItem}" itemLabel="One" itemValue="1"/>
    <!-- More items here. -->
</h:selectOneListbox>
```

A list item which attempts to retrieve data from a managed bean,
but has hardcoded default values if no data is available.

In the managed bean the item, which must be of the type *javax.faces.model.SelectItem,* is initialized in like this:

```
private SelectItem mMenuTwoItem;

@PostConstruct
public void initialize()
{
    // Other initialization...

    mMenuTwoItem = new SelectItem("1", "Item One");
}
```

Initializing the list item data in the managed bean for the element in the above list.

Multiple Items

Multiple, non-hardcoded, items can be supplied using the <f:selectItems> tag. The value expression in the *value* attribute can refer to data in one of four formats:

- A single *javax.faces.model.SelectItem* object.
- An array of *javax.faces.model.SelectItem* objects.
- A *java.util.Collection* of *javax.faces.model.SelectItem* objects.
- A *java.util.Map* containing key-value pairs where the key is converted to a string and used as label for the item and the value, also converted to a string, used as value.

The following example shows how a map is used to provide the items of a list.
First, the code fragment in the web page:

```
<h:outputText value="Select a country:"/>
<h:selectOneMenu id="sel_menu6" value="#{mybean.menuSixSelection}">
    <f:selectItems value="#{mybean.countriesMap}"/>
</h:selectOneMenu>
```

Creating a menu using a map to set the items in the menu.

The map in the managed bean is initialized in the following way. Note that the *java.util.LinkedHashMap* is used. The reason for this is that it preserves the ordering of the items.

```
private Map<String, Object> mCountriesMap;
private String mMenuSixSelection;

@PostConstruct
public void initialize()
{
    mCountriesMap = new LinkedHashMap<String, Object>();
    mCountriesMap.put("Sweden", "se");
    mCountriesMap.put("Germany", "de");
    mCountriesMap.put("England", "uk");
    mCountriesMap.put("Taiwan", "tw");
}
```

Creating and initializing the map of items for display in a list or menu.

The result is a pop-up menu:

Menu using a map for its items, as it appears on the web page.

The *menuSixSelection* property of the managed bean *mybean* will contain one of the values “se”, “de”, “uk” or “tw”.

Grouping Multiple Items

SelectItem objects may also be grouped using the *javax.faces.model.SelectItemGroup* class.
The code in the web page looks similar to what we have seen before.

```
<h:outputText value="Select one or more dishes and/or a beverages:"/>
<h:selectManyListbox id="sel_menu5" value="#{mybean.menuFiveSelection}">
    <f:selectItems value="#{mybean.menuItems}"/>
</h:selectManyListbox>
```

Creating a list with grouped items.

The initialization code in the managed bean is slightly more complex than before. Note that only the array *mMenuItems* need to be an accessible property of the managed bean.

```
private String[] mMenuFiveSelection;
private SelectItem[] mMenuItems;

    @PostConstruct
    public void initialize()
    {
        SelectItem[] theDishesArray;
        SelectItem[] theBeveragesArray;
        SelectItemGroup theDishesGroup;
        SelectItemGroup theBeveragesGroup;

        theDishesArray = new SelectItem[3];
        theDishesArray[0] = new SelectItem("Green Salad");
        theDishesArray[1] = new SelectItem("Lentil Soup");
        theDishesArray[2] = new SelectItem("Baked Potatoes");
        theDishesGroup = new SelectItemGroup("Dishes", "Available dishes",
            false, theDishesArray);

        theBeveragesArray = new SelectItem[3];
        theBeveragesArray[0] = new SelectItem("Water");
        theBeveragesArray[1] = new SelectItem("Orange Juice");
        theBeveragesArray[2] = new SelectItem("Beer");
        theBeveragesGroup = new SelectItemGroup("Beverages",
            "Available beverages", false, theBeveragesArray);

        mMenuItems = new SelectItem[2];
        mMenuItems[0] = theDishesGroup;
        mMenuItems[1] = theBeveragesGroup;
    }
```

Creating and initializing grouped items for display in a list or menu.

Select one or more dishes and/or a beverages:

List with grouped items, as it appears on the web page.

Glossary

Backing Bean
A managed bean with the special tasks of gathering values from UI components and implementing event listener methods. May optionally hold references to user-interface components.

Converter
Converts the value of an UI component into a string and vice versa. An UI component can be associated with a single converter.

Facet
Subordinate UI component with a special relationship to its parent. The subordinate component is not a child of its parent. All facet relations have an unique facet name. Examples are headers and footers of tables.

Internationalization
Also commonly known as “i18n”.
Adapting a piece of software so that its messages are easily translatable and it supports different character sets.

Localization
Also commonly known as “L10n”.
Adapting a piece of software for a specific country or region by translating messages into the language of the country/region and, if necessary, make sure the character set(s) used in the country/region are supported.

Managed Bean
A JavaBean exposed by JSF to EL expressions. Also see Backing Bean. A managed bean is not necessarily a backing bean.

Message
Information or error message generated by an UI component; for instance a message generated by a validator when encountering an illegal value.

Method Expression
A method expression is an expression that specified a method on an object which, at the time of evaluation, cause the method to be invoked. For example, “#{myObject.doSomething}” will cause the method *doSomething* to be invoked on the object referred to by *myObject*.

Renderer
Code separated from corresponding UI component(s), for instance in one or more classes, that is responsible for encoding and decoding one or more types of UI components for a specific type of output, for instance HTML. A single UI component may be associated with several different renderers.

Render Kit
A collection of renderers for a specific type of output, for instance HTML.

UI Component
Stateful object that defines the presentation-independent behaviour of an user-interface element, such as a button or text field but also includes composite components. An UI component may hold data that is related to the presentation of the component, for instance colour or style. Implemented as JavaBeans, with properties, methods and events.

Validator
Responsible for ensuring that data entered into an UI component has a permitted value. An UI component may be associated with multiple validators.

Value Expression
An expression, for instance “This is a string” or “#{myBean.property}”, that is used to assign a value to an attribute. In the former case, the value is constant, while, in the latter case, the value is dynamically retrieved when, for instance, the UI component to which the attribute belongs is to be rendered.
A value expression of the latter kind can also be used to refer to an attribute in a managed bean in which some user-provided data is to be stored.

Appendix 1 – Events and Request Processing Life Cycle Sample Program

This appendix presents a simple web application that was developed in order to visualize the JSF request processing life cycle and how different kinds of events etc. fits into this life cycle.
The application consists of four parts; a web-page, a managed bean, the faces-config.xml file and the web application deployment descriptor web.xml.

Web Page

The web page is minimalistic and consists of two labels, an input field and a button.

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root version="2.0"
          xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html">

    <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>
    <jsp:output omit-xml-declaration="no"
         doctype-root-element="html"
         doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
         doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

    <f:view>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title>JSF Request Processing Life Cycle</title>
            </head>
            <body>
                <h:form>
                    <h3>
                        <h:outputText value="Enter A String"/>
                    </h3>
                    <table>
                        <tr>
                            <td>
                                <h:outputText value="Input Field:"/>
                            </td>
                            <td>
                                <h:inputText value="#{bean.dataString}"
                                             required="true"
                                             validator="#{bean.validateString}"
                                             valueChangeListener=
                                                    "#{bean.stringValueChange}"/>
                            </td>
                        </tr>
                        <tr>
                            <td>
                                <h:commandButton value="Commit Request"
                                                 actionListener="#{bean.doCommit}"/>
                            </td>
                        </tr>
                    </table>
                </h:form>
            </body>
        </html>
    </f:view>
</jsp:root>
```

The source-code for the web page of the example program, index.jspx.

The web page looks like this when being the web application is being run.

Web page of the sample application.

Managed Bean Class

Next is the managed bean class, which contains an attribute holding the data from the input field and a number of methods for different purposes.

```
/**
 * A managed bean that also serves as action listener and phase listener.
 *
 * @author Ivan A Krizsan
 */
public class ManagedBean implements ActionListener, PhaseListener
{
    /* Instance variable(s): */
    /** Holder of string data. */
    private String mDataString;

    /**
     * Initializes the bean and sets it up to listen to phase events
     * and to serve as standard action listener.
     */
    @PostConstruct
    public void initialize()
    {
        FacesContext theContext = FacesContext.getCurrentInstance();
        Application theApp = FacesContext.getCurrentInstance().getApplication();

        System.out.println("*** Initializing managed bean...");
        theApp.setActionListener(this);

        theContext.getViewRoot().addPhaseListener(this);
    }

    /**
     * Cleans up when bean is about to be taken out of service.
     */
    @PreDestroy
    public void cleanup()
    {
        System.out.println("*** Cleaning up managed bean...");
    }

    /**
     * Validates the string data.
     *
     * @param inContext Context.
     * @param inComponent Component holding the user name.
     * @param inValue User name value.
     */
    public void validateString(FacesContext inContext, UIComponent inComponent,
        Object inValue)
    {
        System.out.println("    Validating the string: " + inValue);
    }

    /**
     * Handles events caused by changes in the string field value.
     *
     * @param inEvent Change event.
     */
    public void stringValueChange(ValueChangeEvent inEvent)
    {
```

```
        System.out.println("    Text-field value changed event received.");
        System.out.println("        Value changed from: " +
            inEvent.getOldValue() + " to " + inEvent.getNewValue());
    }

    /**
     * Receives action events associated with the button on the webpage.
     *
     * @param inAction Action event.
     */
    public void doCommit(ActionEvent inAction)
    {
        System.out.println("    Commit button action listener called.");
    }

    /**
     * Handles action events sent to the default action listener.
     *
     * @param inEvent Action event,
     * @throws AbortProcessingException If the processing of the current
     * event is to be aborted.
     */
    public void processAction(ActionEvent inEvent) throws AbortProcessingException
    {
        System.out.println("    Default action listener called.");
    }

    /**
     * Receives notifications that the phase this listener is listening
     * for has ended.
     *
     * @param inEvent Phase event.
     */
    public void afterPhase(PhaseEvent inEvent)
    {
        System.out.println("*** Phase has ended: " + inEvent.getPhaseId());
    }

    /**
     * Receives notifications that the phase this listener is listening
     * for is about to start.
     *
     * @param inEvent Phase event.
     */
    public void beforePhase(PhaseEvent inEvent)
    {
        System.out.println("*** Phase is about to start: " +
            inEvent.getPhaseId());
    }

    /**
     * Returns the id of the phase this listener wants to receive
     * phase events for.
     *
     * @return Id of phase for which to get events.
     */
    public PhaseId getPhaseId()
    {
        return PhaseId.ANY_PHASE;
    }

    /**
     * Retrieves the string entered by the user in the text-field.
     *
     * @return Text-field string.
     */
    public String getDataString()
    {
        System.out.println("    Text-field string retrieved.");
        return mDataString;
    }

    /**
     * Sets the string entered by the user entered in the text-field.
     *
     * @param inDataString Text-field string.
     */
```

```
    public void setDataString(String inDataString)
    {
        System.out.println("    Text-field string set.");
        mDataString = inDataString;
    }
}
```

Source code for the managed bean class of the sample web application.

JSF Configuration File

The JSF configuration file does not offer any surprises – it just configures the managed bean used by the web page.

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

<!-- ========== FULL CONFIGURATION FILE ================================== -->
<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">
    <managed-bean>
        <managed-bean-name>bean</managed-bean-name>
        <managed-bean-class>com.ivan.ManagedBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
</faces-config>
```

JSF configuration file faces-config.xml.

Web Application Deployment Descriptor

Finally, the web.xml file. The only part I have modified in this file is the welcome-file list.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
web-app_2_5.xsd">
    <context-param>
        <param-name>com.sun.faces.verifyObjects</param-name>
        <param-value>false</param-value>
    </context-param>
    <context-param>
        <param-name>com.sun.faces.validateXml</param-name>
        <param-value>true</param-value>
    </context-param>
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>client</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>faces/index.jspx</welcome-file>
    </welcome-file-list>
</web-app>
```

Web application deployment descriptor file, web.xml.

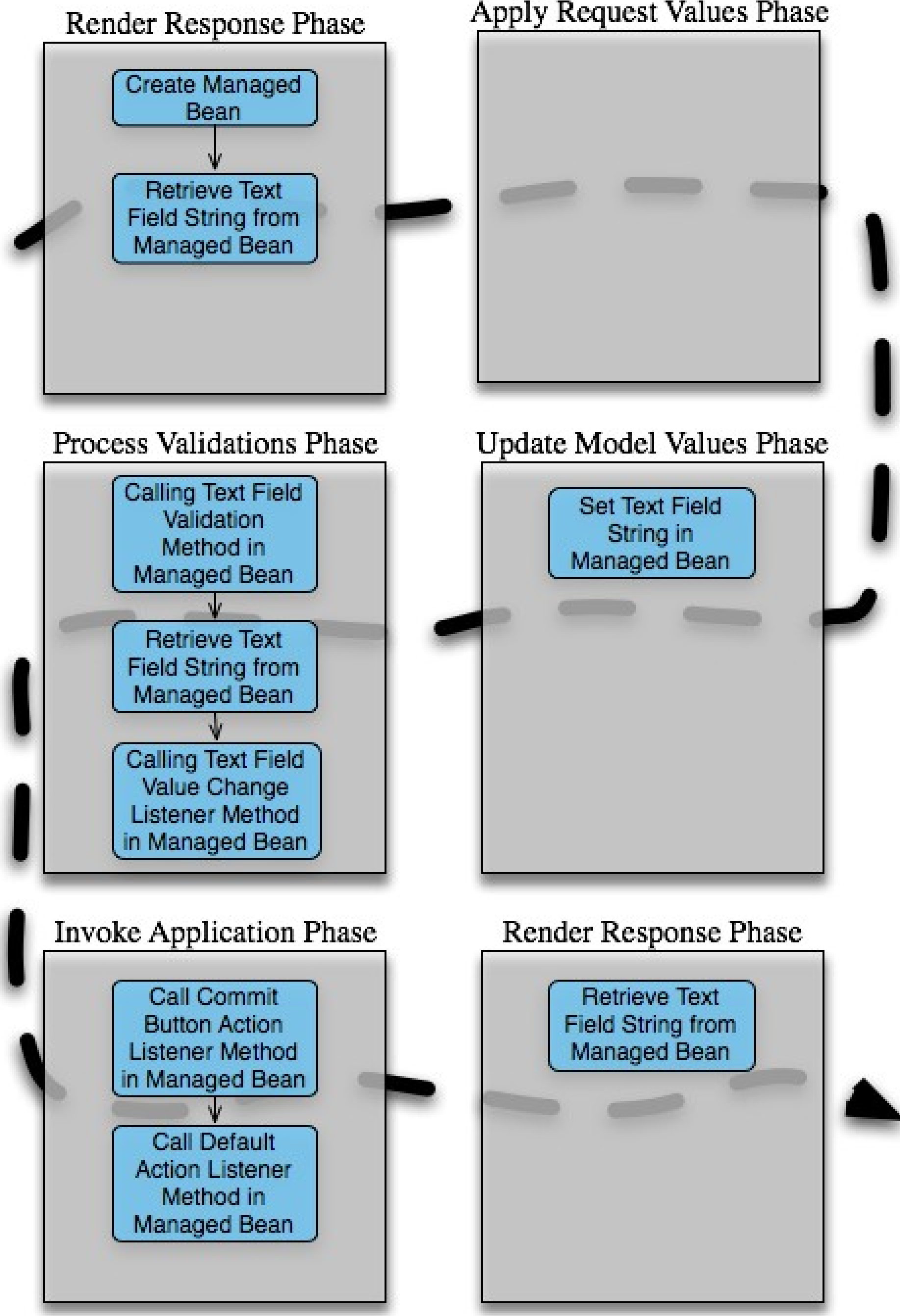
Analysis

Run the web application, enter a string in the text field and click the Commit Request button. Output similar to the following will be written to the web container's console:

```
*** Initializing managed bean...
    Text-field string retrieved.
*** Phase has ended: RENDER_RESPONSE 6
*** Phase is about to start: APPLY_REQUEST_VALUES 2
*** Phase has ended: APPLY_REQUEST_VALUES 2
*** Phase is about to start: PROCESS_VALIDATIONS 3
    Validating the string: test-string
    Text-field string retrieved.
    Text-field value changed event received.
        Value changed from: null to test-string
*** Phase has ended: PROCESS_VALIDATIONS 3
*** Phase is about to start: UPDATE_MODEL_VALUES 4
    Text-field string set.
*** Phase has ended: UPDATE_MODEL_VALUES 4
*** Phase is about to start: INVOKE_APPLICATION 5
    Commit button action listener called.
    Default action listener called.
*** Phase has ended: INVOKE_APPLICATION 5
*** Phase is about to start: RENDER_RESPONSE 6
    Text-field string retrieved.
*** Phase has ended: RENDER_RESPONSE 6
```

Console output from the sample web application.

Looking at a picture showing the different phases and what parts of the sample program that execute in which phase makes things more clear. Note that since the Restore View phase wasn't included in the above output, it has also been left out from the figure.



Phases receiving phase event notification and associated events in the sample program.

Earlier Validation

By adding a single attribute to a UI component, the validation and associated activities can be moved from the Process Validations Phase to the Apply Request Values Phase.
The following part of the web page is changed to include the *immediate* attribute.

```
<td>
    <h:inputText value="#{bean.dataString}"
                 immediate="true"
                 required="true"
                 validator="#{bean.validateString}"
                 valueChangeListener="#{bean.stringValueChange}"/>
</td>
```

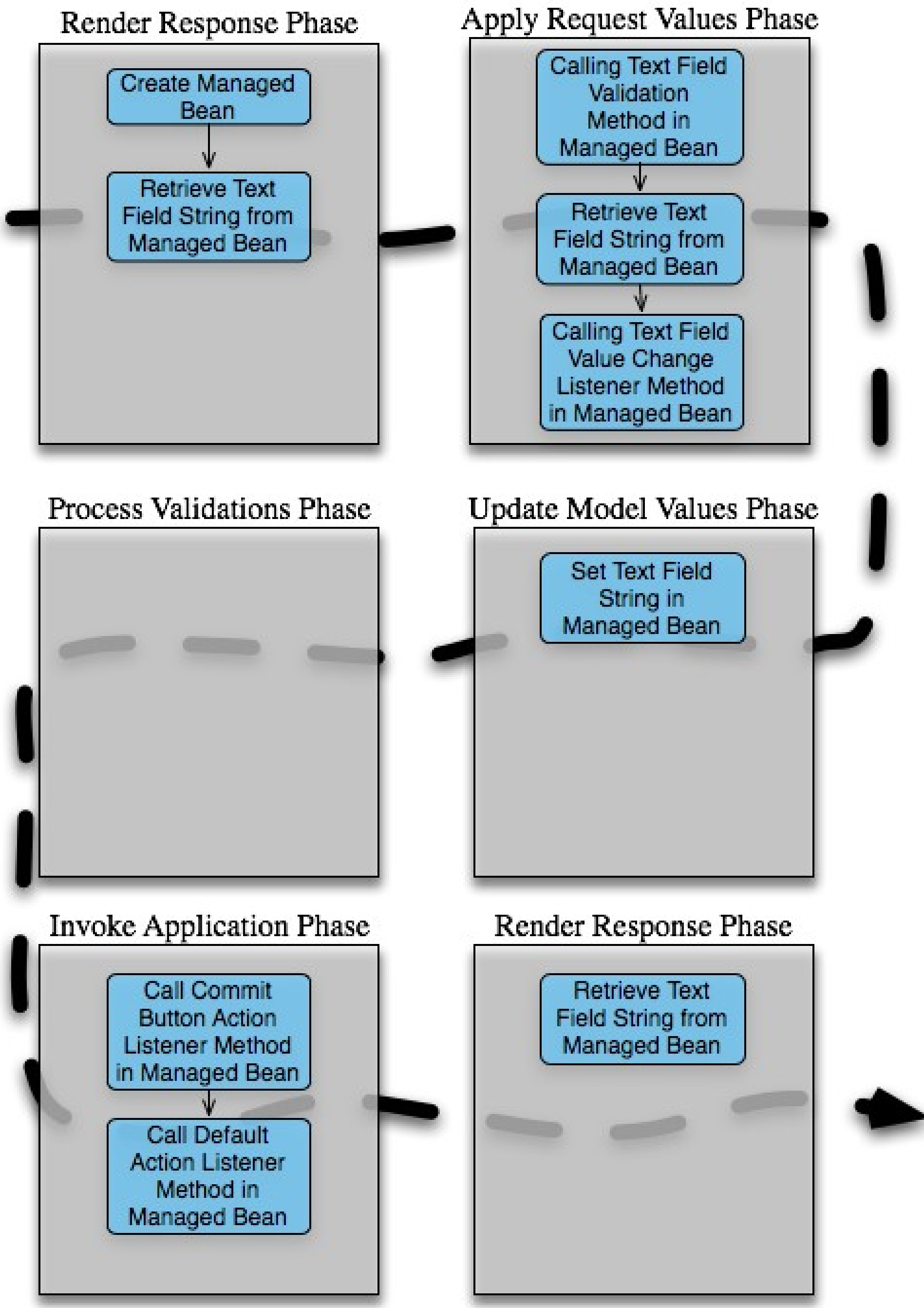
Input text field in the web page with the *immediate* attribute added.

When running the web application with the above change applied, the output to the console is slightly different.

```
*** Initializing managed bean...
    Text-field string retrieved.
*** Phase has ended: RENDER_RESPONSE 6
*** Phase is about to start: APPLY_REQUEST_VALUES 2
    Validating the string: immediate-test
    Text-field string retrieved.
    Text-field value changed event received.
        Value changed from: null to immediate-test
*** Phase has ended: APPLY_REQUEST_VALUES 2
*** Phase is about to start: PROCESS_VALIDATIONS 3
*** Phase has ended: PROCESS_VALIDATIONS 3
*** Phase is about to start: UPDATE_MODEL_VALUES 4
    Text-field string set.
*** Phase has ended: UPDATE_MODEL_VALUES 4
*** Phase is about to start: INVOKE_APPLICATION 5
    Commit button action listener called.
    Default action listener called.
*** Phase has ended: INVOKE_APPLICATION 5
*** Phase is about to start: RENDER_RESPONSE 6
    Text-field string retrieved.
*** Phase has ended: RENDER_RESPONSE 6
```

Console output from the sample web application after having added the *immediate* attribute.

Looking at the corresponding picture showing the different phases and what parts of the sample program that execute in which phase, it is obvious that the events and actions related to the input text field has moved to an earlier phase in the Request Processing Life Cycle. Note that since the Restore View phase wasn't included in the above output, it has also been left out from the figure.



Phases receiving phase event notification and associated events in the sample program after having added the *immediate* attribute to the input text field.

Skipping Phases

Next, some code is added to *validateString* in the managed bean class, which is the method that validates the contents of the input text field. The Request Processing Life Cycle is once again observed.

```
    /**
     * Validates the string data.
     *
     * @param inContext Context.
     * @param inComponent Component holding the user name.
     * @param inValue User name value.
     */
    public void validateString(FacesContext inContext, UIComponent inComponent,
        Object inValue)
    {
        System.out.println("    Validating the string: " + inValue);
        if ("skip".equalsIgnoreCase(((String)inValue)))
        {
            FacesContext.getCurrentInstance().renderResponse();
        }
    }
```

The modified *validateString* method in the managed bean class.

When running the program, first enter “test” in the input field and click the button. Then enter “skip” and click the button.

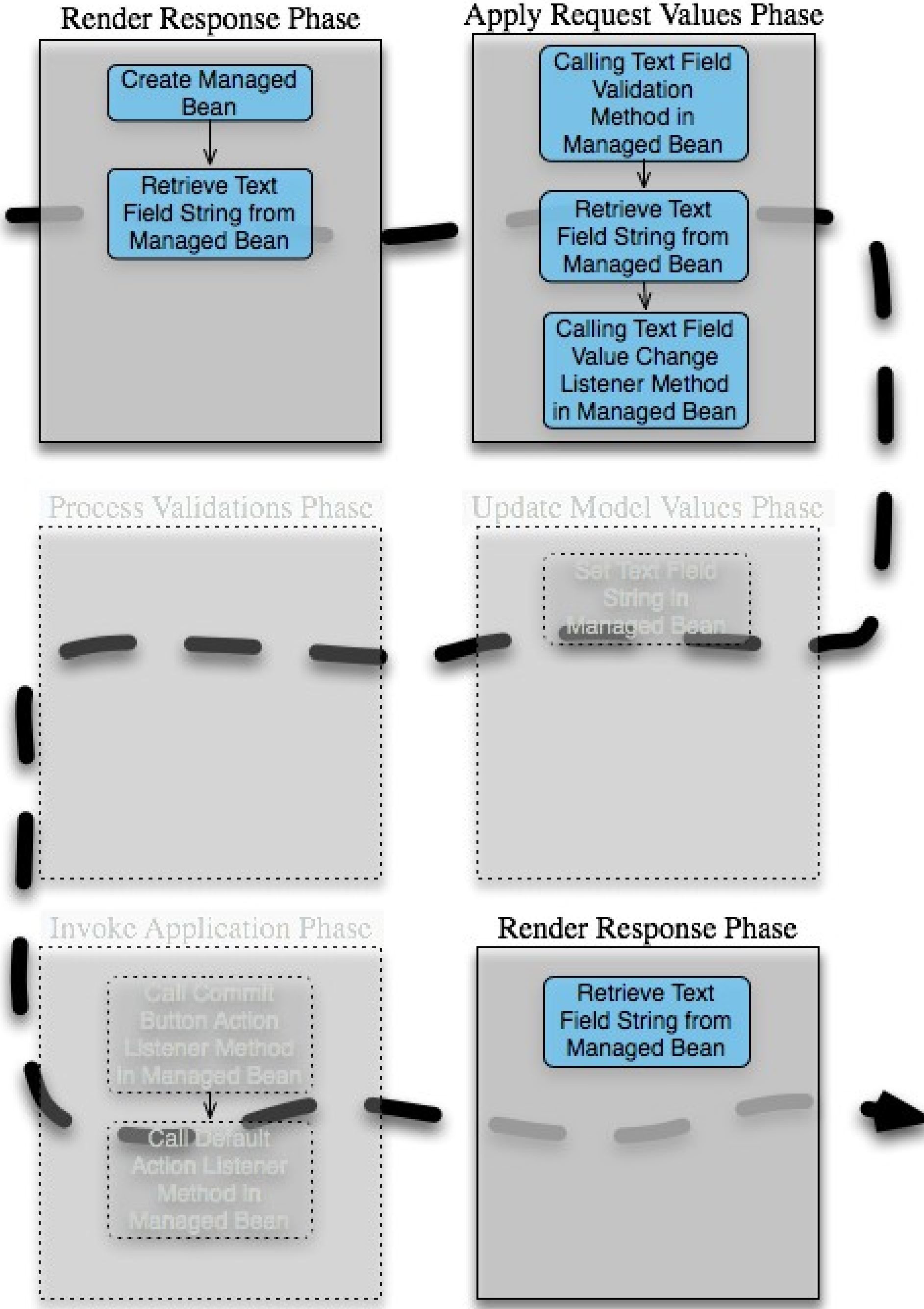
```
*** Initializing managed bean...
    Text-field string retrieved.
*** Phase has ended: RENDER_RESPONSE 6
*** Phase is about to start: APPLY_REQUEST_VALUES 2
    Validating the string: test
    Text-field string retrieved.
    Text-field value changed event received.
        Value changed from: null to test
*** Phase has ended: APPLY_REQUEST_VALUES 2
*** Phase is about to start: PROCESS_VALIDATIONS 3
*** Phase has ended: PROCESS_VALIDATIONS 3
*** Phase is about to start: UPDATE_MODEL_VALUES 4
    Text-field string set.
*** Phase has ended: UPDATE_MODEL_VALUES 4
*** Phase is about to start: INVOKE_APPLICATION 5
    Commit button action listener called.
    Default action listener called.
*** Phase has ended: INVOKE_APPLICATION 5
*** Phase is about to start: RENDER_RESPONSE 6
    Text-field string retrieved.
*** Phase has ended: RENDER_RESPONSE 6

*** Phase is about to start: APPLY_REQUEST_VALUES 2
    Validating the string: skip
    Text-field string retrieved.
    Text-field value changed event received.
        Value changed from: test to skip
*** Phase has ended: APPLY_REQUEST_VALUES 2
*** Phase is about to start: RENDER_RESPONSE 6
*** Phase has ended: RENDER_RESPONSE 6
```

Console output from the sample web application after having modified
the input field validation method in the managed bean class.

Notice that several phases have been skipped the second time, when “skip” was entered in the text field. This is due to the call to the *FacesContext.renderResponse()* method.

The corresponding figure looks like this:



Phases receiving phase event notification and associated events in the sample program when the input text field calls *FacesContext.renderResponse()*.

Bailing Out

Finally, a little more code is added to the *validateString* method in the managed bean class. Another JSP file was also created and named “alternate.jsp”. It is not shown, since it only contains a “Hello World” message.

```
    /**
     * Validates the string data.
     *
     * @param inContext Context.
     * @param inComponent Component holding the user name.
     * @param inValue User name value.
     */
    public void validateString(FacesContext inContext, UIComponent inComponent,
        Object inValue)
    {
        System.out.println("    Validating the string: " + inValue);
        if ("skip".equalsIgnoreCase(((String)inValue)))
        {
            FacesContext.getCurrentInstance().renderResponse();
        }
        if ("end".equalsIgnoreCase((String)inValue))
        {
            try
            {
                FacesContext.getCurrentInstance().getExternalContext().
                    dispatch("/alternate.jsp");
            } catch (IOException theException)
            {
            }
            FacesContext.getCurrentInstance().responseComplete();
        }
    }
```

The modified *validateString* method in the managed bean class.

When running the program, first enter “test” in the input field and click the button, next enter “skip” and click the button and finally enter “end” and click the button.

The console output below has been partitioned to show the respective life cycle for each of the three requests.

```
*** Initializing managed bean...
    Text-field string retrieved.
*** Phase has ended: RENDER_RESPONSE 6
*** Phase is about to start: APPLY_REQUEST_VALUES 2
    Validating the string: test
    Text-field string retrieved.
    Text-field value changed event received.
        Value changed from: null to test
*** Phase has ended: APPLY_REQUEST_VALUES 2
*** Phase is about to start: PROCESS_VALIDATIONS 3
*** Phase has ended: PROCESS_VALIDATIONS 3
*** Phase is about to start: UPDATE_MODEL_VALUES 4
    Text-field string set.
*** Phase has ended: UPDATE_MODEL_VALUES 4
*** Phase is about to start: INVOKE_APPLICATION 5
    Commit button action listener called.
    Default action listener called.
*** Phase has ended: INVOKE_APPLICATION 5
*** Phase is about to start: RENDER_RESPONSE 6
    Text-field string retrieved.
*** Phase has ended: RENDER_RESPONSE 6

*** Phase is about to start: APPLY_REQUEST_VALUES 2
    Validating the string: skip
    Text-field string retrieved.
    Text-field value changed event received.
        Value changed from: test to skip
*** Phase has ended: APPLY_REQUEST_VALUES 2
*** Phase is about to start: RENDER_RESPONSE 6
*** Phase has ended: RENDER_RESPONSE 6

*** Phase is about to start: APPLY_REQUEST_VALUES 2
    Validating the string: end
    Text-field value changed event received.
        Value changed from: skip to end
*** Phase has ended: APPLY_REQUEST_VALUES 2
```

Console output from the sample web application after having modified
the input field validation method in the managed bean class a second time.

Notice that, for the third request when “end” was entered, the Request Processing Life Cycle only entered the Apply Request Values phase and then ended. This is caused by the call to *FacesContext.responseComplete()*.
Also notice that forwarding to another resource is done by obtaining an “external context” from the current *FacesContext*. The external context is the connection to the application environment that JSF is contained in, normally a servlet or portlet environment.

Appendix 2 – UI Component Tree Navigation

This appendix contains a example program that looks closer at UI component identifiers and how UI components in a JSF view can be located using these identifiers.
The application consists of six parts; a web-page, a header JSP file, a footer JSP file, a managed bean, the faces-config.xml file and the web application deployment descriptor web.xml.

Web Page

Look at the code for the *index.jsp* web page in the example program and note the UI components that has been assigned an id and the components that has not.

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<f:view>
    <html>
        <head>
            <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
            <title>JSF Component Identifiers</title>
        </head>
        <body>
            <h:form id="form" binding="#{viewWorm.formComponent}">
                <h1><h:outputText value="JSF Component IDs" /></h1>

                <h:outputLink id="output_link" value="page2.html" title="Go to page2">
                        <h3>Page 2</h3>
                </h:outputLink>

                <c:forEach begin="1" end="3”>
                    <h:outputText id="loop_text" value="Many times..."/>
                    <br/>
                </c:forEach>

                <h:panelGrid id="panel_grid" columns="4" border="1">
                    <f:facet name="header">
                        <f:subview id="header">
                            <c:import url="header.jsp"/>
                        </f:subview>
                    </f:facet>

                    <h:outputText id="cell_1" value="Cell 1"/>
                    <h:outputText id="cell_2" value="Cell 2"/>
                    <h:outputText id="cell_3" value="Cell 3"/>
                    <h:outputText id="cell_4" value="Cell 4"/>
                    <h:outputText id="cell_5" value="Cell 5"/>
                    <h:outputText id="cell_6" value="Cell 6"/>
                    <h:outputText id="cell_7" value="Cell 7"/>

                    <f:facet name="footer">
                        <f:subview id="footer">
                            <c:import url="footer.jsp"/>
                        </f:subview>
                    </f:facet>
                </h:panelGrid>
                <br/><br/>
                <h:commandButton id="command_button" value="Click Me"
                   action="#{viewWorm.doButton}"/>
            </h:form>
        </body>
    </html>
</f:view>
```

The source-code for the web page of the sample program, index.jsp.

A few things should be noted about the web page:

- The <f:view> tag has not been assigned an identifier.
- The <h:form> is bound to a property in the managed bean.
- Even though there is a link to a page2.html, the example does not include such a file.
- The <h:outputText> tag inside the loop has the id “loop_text”.
- The <c:import> JSTL tags in the panel grid header and footer have been wrapped in <f:subview> tags.

The header and footer JSP files “header,jsp” and “footer.jsp” are almost identical. Both these files reference an image file “metal.jsp” which is not included here, but can be replaced with any small image file.

```
<%-- this is header.jsp --%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<h:panelGrid id="grid" columns="2">
    <h:graphicImage id="image" value="metal.jpg"/>
    <h:outputText id="text" value="Header Title"/>
</h:panelGrid>
```

Main page panel grid header file, header.jsp.

```
<%-- this is footer.jsp --%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<h:panelGrid id="grid" columns="2">
    <h:graphicImage id="image" value="metal.jpg"/>
    <h:outputText id="text" value="Footer Title"/>
</h:panelGrid>
```

Main page panel grid footer file, footer.jsp.

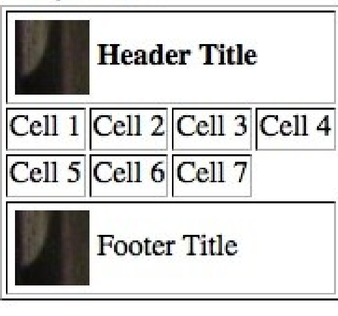
Note that the the three tags in the header has the same identifiers as the three tags in the footer,

The web page produced by the above three files will look something like this:

Subviews in JSF

Page 2

Many times...
Many times...
Many times...



Click Me

Component tree navigation example program web page.

Managed Bean Class

The managed bean class contains a UI component property, the one that the <h:form> tag in the web page is bound to, as well as the action method invoked when the button is clicked and some support methods.

```
package com.ivan;

import java.util.List;
import java.util.Map;
import javax.faces.component.UIComponent;

/**
 * Managed bean class that prints JSF UI component identifiers
 * starting from the parent of the component bound to the bean.
 *
 * @author Ivan A Krizsan
 */
public class ViewWorm
{
    /* Constant(s): */

    /* Instance variable(s): */
    /** Form component on web page. */
    private UIComponent mFormComponent;

    /**
     * Handles button clicks on the web page.
     *
     * @return Ressult of processing, always successful.
     */
    public String doButton()
    {
        int theIndent = 1;
        UIComponent theRoot;
        UIComponent theFoundComponent;

        /*
         * There is always at least a <f:view> parent, since
         * the <f:view> cannot be bound to anything.
         */
        theRoot = mFormComponent.getParent();
        System.out.println("Root family: " + theRoot.getFamily() +
            ", id: " + theRoot.getId());

        printChildrenIds(theRoot, theIndent);

        /* Find some components. */
        theFoundComponent = theRoot.findComponent(":form:command_button");
        System.out.println("Finding component: :form:command_button");
        System.out.println(" Found component: " + theFoundComponent);

        theFoundComponent = theRoot.findComponent("form:header:image");
        System.out.println("Finding component: form:header:image");
        System.out.println(" Found component: " + theFoundComponent);

        theFoundComponent = theRoot.findComponent(":form");
        System.out.println("Finding component: :form");
        System.out.println(" Found component: " + theFoundComponent);

        theFoundComponent = theFoundComponent.findComponent("header:image");
        System.out.println("Finding component: header:image using :form as starting-
point.");
        System.out.println(" Found component: " + theFoundComponent);

        return "success";
    }

    /**
     * Creates a string containing the supplied number of spaces.
     *
     * @param inIndent Number of spaces.
     * @return String of spaces.
     */
    private String createIndentString(final int inIndent)
```

```
    {
        String theIndentStr = new String();

        for (int i = 0; i < inIndent; i++)
        {
            theIndentStr += " ";
        }

        return theIndentStr;
    }

    /**
     * Prints the identifiers of all the child components of the supplied
     * component, indenting the output with supplied number of spaces.
     * For each child component, recursively print the identifiers of all
     * its child components.
     *
     * @param inComponent Component which children to print identifiers of.
     * @param inIndent Number of spaces to indent the output with.
     */
    private void printChildrenIds(UIComponent inComponent, final int inIndent)
    {
        List<UIComponent> theChildren;
        String theIndentStr = createIndentString(inIndent);

        theChildren = inComponent.getChildren();
        for (UIComponent theChild : theChildren)
        {
            System.out.println(theIndentStr + "Child family: " +
                theChild.getFamily() + ", id: " + theChild.getId());
            printFacetIds(theChild, inIndent + 1);
            printChildrenIds(theChild, inIndent + 1);
        }
    }

    /**
     * Prints the identifiers of all facets of the supplied component,
     * indenting the output with supplied number of spaces.
     * For each facet, recursively print the identifiers of all its
     * child components.
     *
     * @param inComponent Component which facet identifiers to print.
     * @param inIndent Number of spaces to indent the output with.
     */
    private void printFacetIds(UIComponent inComponent, final int inIndent)
    {
        Map<String, UIComponent> theFacets;
        String theIndentStr = createIndentString(inIndent);

        theFacets = inComponent.getFacets();
        for (String theFacetName : theFacets.keySet())
        {
            UIComponent theFacet;

            theFacet = theFacets.get(theFacetName);
            System.out.println(theIndentStr + "Facet name: " + theFacetName +
                ", facet id: " + theFacet.getId());
            printChildrenIds(theFacet, inIndent + 1);
        }
    }

    public UIComponent getFormComponent()
    {
        return mFormComponent;
    }

    public void setFormComponent(UIComponent inFormComponent)
    {
        mFormComponent = inFormComponent;
    }
}
```

Source code for the managed bean class, ViewWorm.java, of the sample web application.

JSF Configuration File

The JSF configuration file does not offer any surprises – it just configures the managed bean used by the web page.

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

<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">
    <managed-bean>
        <description>
            Bean that finds IDs of sub-components.
        </description>
        <managed-bean-name>viewWorm</managed-bean-name>
        <managed-bean-class>com.ivan.ViewWorm</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
</faces-config>
```

JSF configuration file faces-config.xml.

Web Application Deployment Descriptor

Finally, the web.xml file. The only part I have modified in this file is the welcome-file list.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
web-app_2_5.xsd">
    <context-param>
        <param-name>com.sun.faces.verifyObjects</param-name>
        <param-value>false</param-value>
    </context-param>
    <context-param>
        <param-name>com.sun.faces.validateXml</param-name>
        <param-value>true</param-value>
    </context-param>
    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>client</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>faces/index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Web application deployment descriptor file, web.xml.

Analysis

After having run the web application and clicked the Click Me button, output similar to the following can be found in the web container's console:

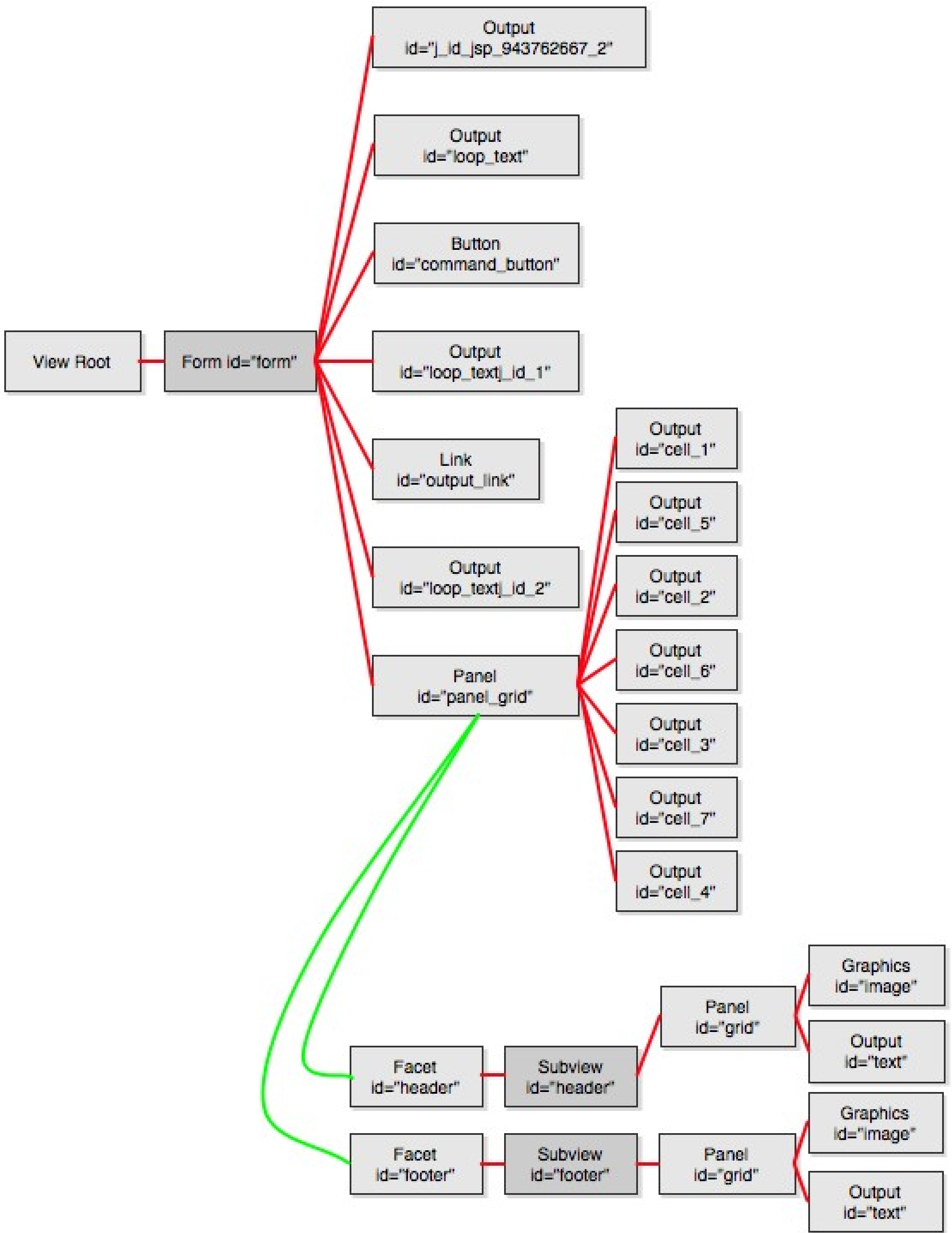
```
Root family: javax.faces.ViewRoot, id: j_id_jsp_1670181326_0
 Child family: javax.faces.Form, id: form
  Child family: javax.faces.Output, id: j_id_jsp_1670181326_2
  Child family: javax.faces.Output, id: output_link
  Child family: javax.faces.Output, id: loop_text
  Child family: javax.faces.Output, id: loop_textj_id_1
  Child family: javax.faces.Output, id: loop_textj_id_2
  Child family: javax.faces.Panel, id: panel_grid
   Facet name: footer, facet id: footer
    Child family: javax.faces.Panel, id: grid
     Child family: javax.faces.Graphic, id: image
     Child family: javax.faces.Output, id: text
   Facet name: header, facet id: header
    Child family: javax.faces.Panel, id: grid
     Child family: javax.faces.Graphic, id: image
     Child family: javax.faces.Output, id: text
   Child family: javax.faces.Output, id: cell_1
   Child family: javax.faces.Output, id: cell_2
   Child family: javax.faces.Output, id: cell_3
   Child family: javax.faces.Output, id: cell_4
   Child family: javax.faces.Output, id: cell_5
   Child family: javax.faces.Output, id: cell_6
   Child family: javax.faces.Output, id: cell_7
  Child family: javax.faces.Command, id: command_button
Finding component: :form:command_button
 Found component: javax.faces.component.html.HtmlCommandButton@fdc7a3
Finding component: form:header:image
 Found component: javax.faces.component.html.HtmlGraphicImage@3db5eb
Finding component: :form
 Found component: javax.faces.component.html.HtmlForm@7290de
Finding component: header:image using :form as starting-point.
 Found component: javax.faces.component.html.HtmlGraphicImage@3db5eb
```

Console output from the sample web application.

Some noteworthy things about the above output:

- The view root, that is the <f:view> tag, has been assigned the id j_id_jsp_1670181326_0.
  This id might be different, depending on the JSF implementation used etc.
- One text output item has been created for each pass through the loop.
  The assigned id has been used in the first pass and, in subsequent passes, a short string has been appended to the assigned id to create a new, unique, id.
- The components in the header and footer of the panel have retained their identifiers.
  This is because of having wrapped the <c:import> JSTL tags in the panel grid header and footer with <f:subview> tags. The <f:subview> tags are invisible in the above output, but creates two naming containers which allows the header and footer to contain components with the same identifiers without resulting in a conflict.
- The search for the component ":form:header:image" starting from the view component and the search for the component "header:image" starting from the form component yields the same result.
  This shows the use of absolute versus relative search expressions.

Comparing the above output with the component tree for the web page, things might become a little more clear. Red lines are used to denote parent-child relationships, green lines are used to denote facets relationships. UI components with a darker colour are naming containers.



Component tree hierarchy for the sample program web page.

If the <f:subview> tags are removed from the web page and the sample web application is run, the following occurs:

```
HTTP Status 500 -
type Exception report
message
description The server encountered an internal error () that prevented it from
fulfilling this request.
exception
org.apache.jasper.JasperException: javax.servlet.ServletException:
javax.servlet.jsp.JspException: javax.servlet.ServletException:
javax.servlet.jsp.JspException: java.lang.IllegalStateException: Duplicate component id:
'form:grid', first used in tag: 'com.sun.faces.taglib.html_basic.PanelGridTag'
+id: j_id_jsp_1670181326_0
...
```

Trying to run the sample program without <f:subview> tags produces an error.

This error occurs, as the description says, due to duplicate component ids. The identifier “grid”, found in both the header and the footer, occurs twice within the same naming container.

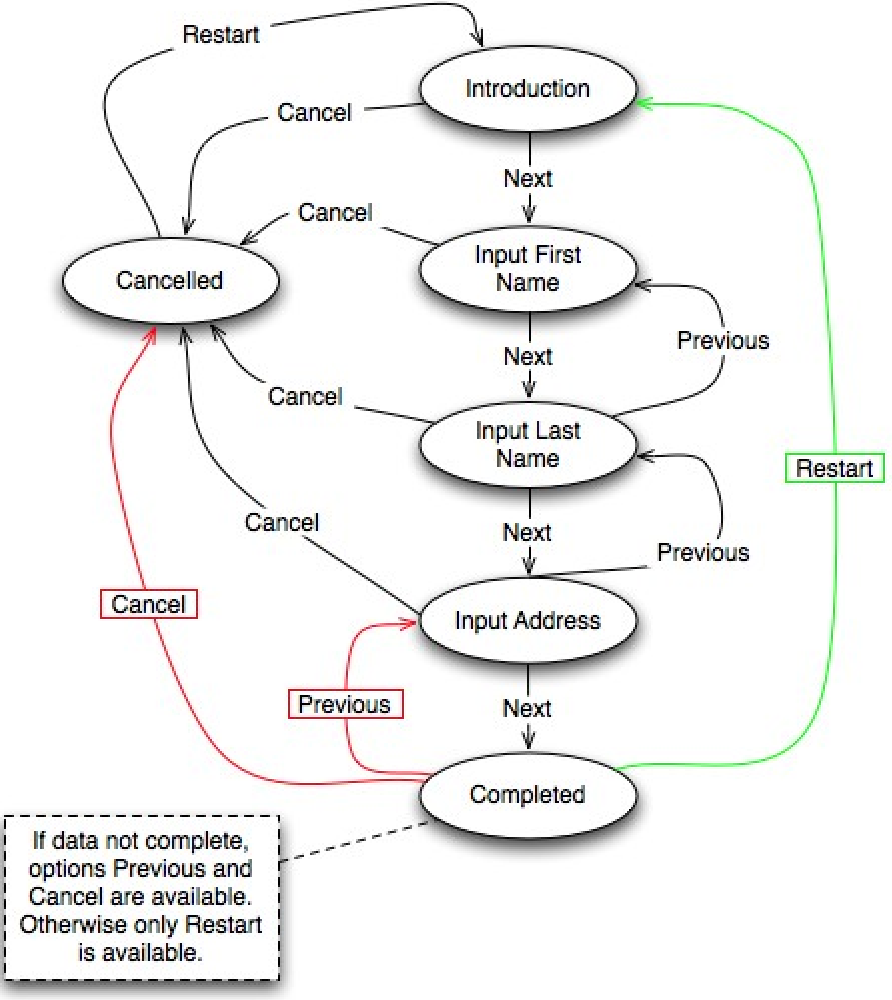
Appendix 3 – Navigation Sample Program

This appendix contains a sample program that will show the following things:

- How to configure navigation between a number of JSF views.
- How to separate the JSF configuration file, faces-config.xml, into multiple files. For instance one configuration file for navigation, one for managed beans etc.
- How to use JavaScript to give an input component focus.

Navigation Schema

The purpose of the application is to ask the user to input his/her first and last names and address. This is done in a number of steps, as shown in the following figure:



Navigational paths between the different views in the navigation sample program.

Before starting to write a JSF user interface that consists of multiple pages, drawing a figure like the one above is a good idea since it clarifies how navigation is to be configured.
UML activity diagrams may also be a good alternative, as described by Benjamin Lieberman in the article UML Activity Diagrams: Detailing User Interface Navigation.

Java Script

The following JavaScript function is used by three of the web pages below to give the appropriate input field focus. It is stored in the file "focus.js" in the same directory as the web pages.

```
function focus(inComponentId)
{
    var element = document.getElementById(inComponentId);
    if (element != null)
    {
        element.focus();
    }
}
```

JavaScript focus function.

Web Pages

Here, the code for all the web pages of the sample program is listed. A smaller font size has been chosen, in order to conserve space in this document. Copy the contents to a text editor for better reading conditions.

cancelled.jspx

The web page displayed when the user cancels the data input.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Document   : cancelled
Author     : Ivan A Krizsan
-->
<jsp:root version="2.0"
          xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html">

    <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>
    <jsp:output omit-xml-declaration="no"
         doctype-root-element="html"
         doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
         doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

    <f:view>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title>Cancelled</title>
            </head>
            <body>
                <h:form>
                    <p>
                        <h:outputText value="The questionarie has been cancelled!"/>
                        <br/>
                        <h:outputText value="Press Restart to start over."/>
                    </p>
                    <p>
                        <h:commandButton value="Restart"
                                         action="#{cancelledController.doRestart}"/>
                    </p>
                </h:form>
            </body>
        </html>
    </f:view>
</jsp:root>
```

The code for cancelled.jspx of the sample program.

completed.jspx

The web page shown when the user has completed the data input.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Document   : completed
Author     : Ivan A Krizsan
-->
<jsp:root version="2.0"
          xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:c="http://java.sun.com/jsp/jstl/core">

    <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>
    <jsp:output omit-xml-declaration="no"
         doctype-root-element="html"
         doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
         doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

    <f:view>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title>Completed!</title>
            </head>
            <body>
                <h:form>
                    <p>
                        <c:choose>
                            <c:when test="${completedController.dataComplete}">
                                <h:outputText value="Thank you for providing the data!"/>
                                <br/>
                                <h:outputText value="Click the Restart button to start over."/>
                            </c:when>
                            <c:otherwise>
                                <h:outputText value="Your data is incomplete!"/>
                                <br/>
                                <h:outputText value="Click the Cancel button to end or the Previous button to go back."/>
                            </c:otherwise>
                        </c:choose>
                    </p>

                    <p>
                        <h:panelGrid columns="2">
                            <h:outputLabel value="First name:"/>
                            <h:outputText value="#{personBean.firstName}"/>

                            <h:outputLabel value="Last name:"/>
                            <h:outputText value="#{personBean.lastName}"/>

                            <h:outputLabel value="Address:"/>
                            <h:outputText value="#{personBean.address}"/>
                        </h:panelGrid>
                    </p>

                    <c:choose>
                        <c:when test="${completedController.dataComplete}">
                            <h:commandButton value="Restart" type="submit"
                                             action="#{completedController.doRestart}"/>
                        </c:when>
                        <c:otherwise>
                            <h:panelGrid columns="2">
                        <h:commandButton value="Cancel" type="submit"
                                         action="#{completedController.doCancel}"/>
                        <h:commandButton value="Previous" type="submit"
                                         action="#{completedController.doPrevious}"/>
                    </h:panelGrid>
                        </c:otherwise>
                    </c:choose>
                </h:form>
            </body>
        </html>
    </f:view>
</jsp:root>
```

The code for completed.jspx of the sample program.

input_address.jspx

The web page requesting the user to input his/her address.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Document   : input_address
Author     : Ivan A Krizsan
-->
<jsp:root version="2.0"
          xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html">

    <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>
    <jsp:output omit-xml-declaration="no"
         doctype-root-element="html"
         doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
         doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

    <f:view>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title>Input Address</title>
            </head>

            <script type="text/javascript">
                <jsp:directive.include file="focus.js"/>
            </script>

            <body onload="focus('form:address_input')">
                <h:form id="form">
                    <p>
                        <h:outputText value="Please enter your address!"/>
                    </p>
                    <p>
                        <h:panelGrid columns="2">
                            <h:outputLabel for="firstname_input" value="First name:"/>
                            <h:inputText id="firstname_input"
                                         value="#{personBean.firstName}" disabled="true"/>

                            <h:outputLabel for="lastname_input" value="Last name:"/>
                            <h:inputText id="lastname_input"
                                         value="#{personBean.lastName}" disabled="true"/>

                            <h:outputLabel for="address_input"  value="Address:"/>
                            <h:inputText id="address_input"
                                         value="#{personBean.address}" disabled="false"/>
                        </h:panelGrid>
                    </p>

                    <h:panelGrid columns="3">
                        <h:commandButton value="Next" type="submit"
                                         action="#{inputAddressController.doNext}"/>
                        <h:commandButton value="Cancel" type="submit"
                                         action="#{inputAddressController.doCancel}"/>
                        <h:commandButton value="Previous" type="submit"
                                         action="#{inputAddressController.doPrevious}"/>
                    </h:panelGrid>
                </h:form>
            </body>
        </html>
    </f:view>
</jsp:root>
```

The code for input_address.jspx of the sample program.

input_firstname.jspx

The web page requesting the user to input his/her first name.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Document   : input_firstname
Author     : Ivan A Krizsan
-->
<jsp:root version="2.0"
          xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html">

    <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>
    <jsp:output omit-xml-declaration="no"
         doctype-root-element="html"
         doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
         doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

    <f:view>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title>Input First Name</title>
            </head>

            <script type="text/javascript">
                <jsp:directive.include file="focus.js"/>
            </script>

            <body onload="focus('form:firstname_input')">
                <h:form id="form">
                    <p>
                        <h:outputText value="Please enter your first name!"/>
                    </p>
                    <p>
                        <h:panelGrid columns="2">
                            <h:outputLabel for="firstname_input" value="First name:"/>
                            <h:inputText id="firstname_input"
                                         value="#{personBean.firstName}" disabled="false"/>

                            <h:outputLabel for="lastname_input" value="Last name:"/>
                            <h:inputText id="lastname_input"
                                         value="#{personBean.lastName}" disabled="true"/>

                            <h:outputLabel for="address_input"  value="Address:"/>
                            <h:inputText id="address_input"
                                         value="#{personBean.address}" disabled="true"/>
                        </h:panelGrid>
                    </p>

                    <h:panelGrid columns="2">
                        <h:commandButton value="Next" type="submit"
                                         action="#{inputFirstNameController.doNext}"/>
                        <h:commandButton value="Cancel" type="submit"
                                         action="#{inputFirstNameController.doCancel}"/>
                    </h:panelGrid>
                </h:form>

            </body>
        </html>
    </f:view>
</jsp:root>
```

The code for input_firstname.jspx of the sample program.

input_lastname.jspx

The web page requesting the user to input his/her last name.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Document   : input_lastname
Author     : Ivan A Krizsan
-->
<jsp:root version="2.0"
          xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html">

    <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>
    <jsp:output omit-xml-declaration="no"
         doctype-root-element="html"
         doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
         doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

    <f:view>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title>Input Last Name</title>
            </head>

            <script type="text/javascript">
                <jsp:directive.include file="focus.js"/>
            </script>

            <body onload="focus('form:lastname_input')">
                <h:form id="form">
                    <p>
                        <h:outputText value="Please enter your last name!"/>
                    </p>
                    <p>
                        <h:panelGrid columns="2">
                            <h:outputLabel for="firstname_input" value="First name:"/>
                            <h:inputText id="firstname_input"
                                         value="#{personBean.firstName}" disabled="true"/>

                            <h:outputLabel for="lastname_input" value="Last name:"/>
                            <h:inputText id="lastname_input"
                                         value="#{personBean.lastName}" disabled="false"/>

                            <h:outputLabel for="address_input"  value="Address:"/>
                            <h:inputText id="address_input"
                                         value="#{personBean.address}" disabled="true"/>
                        </h:panelGrid>
                    </p>

                    <h:panelGrid columns="3">
                        <h:commandButton value="Next" type="submit"
                                         action="#{inputLastNameController.doNext}"/>
                        <h:commandButton value="Cancel" type="submit"
                                         action="#{inputLastNameController.doCancel}"/>
                        <h:commandButton value="Previous" type="submit"
                                         action="#{inputLastNameController.doPrevious}"/>
                    </h:panelGrid>
                </h:form>

            </body>
        </html>
    </f:view>
</jsp:root>
```

The code for input_lastname.jspx of the sample program.

introduction.jspx

The web page showing an introduction to the sample application.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Document   : introduction
Author     : Ivan A Krizsan
-->
<jsp:root version="2.0"
          xmlns:jsp="http://java.sun.com/JSP/Page"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html">

    <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>
    <jsp:output omit-xml-declaration="no"
         doctype-root-element="html"
         doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
         doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

    <f:view>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title>The First Page</title>
            </head>
            <body>
                <h:form>
                    <p>
                        <h:outputText value="Welcome!"/>
                    </p>
                    <p>
                        <h:outputText value=
                            "In the following pages, you will be asked to input some data about yourself."/>
                    </p>
                    <h:panelGrid columns="2">
                        <h:commandButton id="continue" value="Continue" type="submit"
                            action="#{introductionController.doContinue}"/>
                        <h:commandButton id="cancel" value="Cancel" type="submit"
                            action="#{introductionController.doCancel}"/>
                    </h:panelGrid>
                </h:form>
            </body>
        </html>
    </f:view>
</jsp:root>
```

The code for introduction.jspx of the sample program.

Managed Bean Classes

The code for the managed bean classes. A smaller font size has been chosen, in order to conserve space in this document. Copy the contents to a text editor for better reading conditions.

BaseController

The base controller class enables sharing of common code between some controller classes.

```
package com.ivan.controllers;

import com.ivan.model.PersonBean;
import java.util.Map;
import javax.faces.context.FacesContext;

/**
 * Base class for view-controllers, enabling sharing of common properties.
 *
 * @author Ivan A Krizsan
 */
public class BaseController
{
    /**
     * Determines if all data has been supplied by the user.
     *
     * @return True if all data has been supplied by user, false otherwise.
     */
    public boolean isDataComplete()
    {
        boolean theCompleteFlag;
        PersonBean thePerson = findPersonBean();

        if (thePerson == null)
        {
            return false;
        }

        theCompleteFlag = (thePerson.getFirstName().length() > 0);
        theCompleteFlag &= (thePerson.getLastName().length() > 0);
        theCompleteFlag &= (thePerson.getAddress().length() > 0);

        return theCompleteFlag;
    }

    /**
     * Clears the first name, last name and address of the current
     * person, if any.
     */
    protected void clearFields()
    {
        PersonBean thePerson = findPersonBean();
        if (thePerson != null)
        {
            thePerson.setFirstName("");
            thePerson.setLastName("");
            thePerson.setAddress("");
        }
    }

    /**
     * Clears the first name of the current person, if any.
     */
    protected void clearFirstName()
    {
        PersonBean thePerson = findPersonBean();
        if (thePerson != null)
        {
            thePerson.setFirstName("");
            thePerson.setLastName("");
            thePerson.setAddress("");
        }
    }

    /**
     * Clears the last name of the current person, if any.
     */
    protected void clearLastName()
    {
        PersonBean thePerson = findPersonBean();
        if (thePerson != null)
        {
            thePerson.setLastName("");
```

```
        }
    }

    /**
     * Clears the address of the current person, if any.
     */
    protected void clearAddress()
    {
        PersonBean thePerson = findPersonBean();
        if (thePerson != null)
        {
            thePerson.setAddress("");
        }
    }

    /**
     * Finds the person bean in the current session.
     *
     * @return Current person bean, or null.
     */
    protected PersonBean findPersonBean()
    {
        Map<String, Object> theSessionAttributes;
        PersonBean thePerson;

        theSessionAttributes = FacesContext.getCurrentInstance().
            getExternalContext().getSessionMap();
        thePerson = (PersonBean) theSessionAttributes.get("personBean");

        return thePerson;
    }
}
```

The base controller class of the sample program.

CancelledController

The controller for the cancelled view of the sample program.

```
package com.ivan.controllers;

/**
 * This class implements the view-controller for the Cancelled page.
 *
 * @author Ivan A Krizsan
 */
public class CancelledController extends BaseController
{
    /**
     * Processes clicks on the Restart button.
     *
     * @return Action result.
     */
    public String doRestart()
    {
        clearFields();

        return "success";
    }
}
```

The cancelled view controller class of the sample program.

CompletedController

The controller for the completed view of the sample program.

```
package com.ivan.controllers;

/**
 * This class implements the view-controller for the Completed page.
 *
 * @author Ivan A Krizsan
 */
public class CompletedController extends BaseController
{
    /**
     * Processes clicks on the Restart button.
     *
     * @return Action result.
     */
    public String doRestart()
    {
        clearFields();

        return "success";
    }

     /**
     * Processes clicks on the Previous button.
     *
     * @return Action result.
     */
    public String doPrevious()
    {
        return "success";
    }

    /**
     * Processes clicks on the Cancel button.
     *
     * @return Action result.
     */
    public String doCancel()
    {
        return "cancel";
    }
}
```

The completed view controller class of the sample program.

InputAddressController

The controller for the input address view of the sample program.

```
package com.ivan.controllers;

/**
 * This class implements the view-controller for the Input Address page.
 *
 * @author Ivan A Krizsan
 * @version Apr 7, 2008
 * @since Apr 2, 2008
 */
public class InputAddressController
{
    /**
     * Processes clicks on the Next button.
     *
     * @return Action result.
     */
    public String doNext()
    {
        return "success";
    }

    /**
     * Processes clicks on the Previous button.
     *
     * @return Action result.
     */
    public String doPrevious()
    {
        return "success";
    }

    /**
     * Processes clicks on the Cancel button.
     *
     * @return Action result.
     */
    public String doCancel()
    {
        return "cancel";
    }
}
```

The input address view controller class of the sample program.

InputFirstNameController

The controller for the input first name view of the sample program.

```
package com.ivan.controllers;

/**
 * This class implements the view-controller for the Input First Name page.
 *
 * @author Ivan A Krizsan
 */
public class InputFirstNameController
{
    /**
     * Processes clicks on the Next button.
     *
     * @return Action result.
     */
    public String doNext()
    {
        return "success";
    }

    /**
     * Processes clicks on the Cancel button.
     *
     * @return Action result.
     */
    public String doCancel()
    {
        return "cancel";
    }
}
```

The input first name view controller class of the sample program.

InputLastNameController

The controller for the input last name view of the sample program.

```
package com.ivan.controllers;

/**
 * This class implements the view-controller for the Input Last Name page.
 *
 * @author Ivan A Krizsan
 */
public class InputLastNameController
{
     /**
     * Processes clicks on the Next button.
     *
     * @return Action result.
     */
    public String doNext()
    {
        return "success";
    }

    /**
     * Processes clicks on the Previous button.
     *
     * @return Action result.
     */
    public String doPrevious()
    {
        return "success";
    }

    /**
     * Processes clicks on the Cancel button.
     *
     * @return Action result.
     */
    public String doCancel()
    {
        return "cancel";
    }
}
```

The input last name view controller class of the sample program.

IntroductionController

The controller for the introduction view of the sample program.

```
package com.ivan.controllers;

/**
 * This class implements the view-controller for the Introduction page.
 *
 * @author Ivan A Krizsan
 */
public class IntroductionController
{
    /**
     * Processes clicks on the Continue button.
     *
     * @return Action result.
     */
    public String doContinue()
    {
        return "success";
    }

    /**
     * Processes clicks on the Cancel button.
     *
     * @return Action result.
     */
    public String doCancel()
    {
        return "cancel";
    }
}
```

The introduction view controller class of the sample program.

PersonBean

Managed bean holding the user-submitted data in the sample program.

```
package com.ivan.model;

/**
 * This class implements a model object holding data about one person.
 *
 * @author Ivan A Krizsan
 */
public class PersonBean
{
    /* Instance Variable(s): */
    private String mFirstName = "";
    private String mLastName = "";
    private String mAddress = "";

    public String getAddress()
    {
        return mAddress;
    }

    public void setAddress(String inAddress)
    {
        mAddress = inAddress;
    }

    public String getFirstName()
    {
        return mFirstName;
    }

    public void setFirstName(String inFirstName)
    {
        mFirstName = inFirstName;
    }

    public String getLastName()
    {
        return mLastName;
    }

    public void setLastName(String inLastName)
    {
        mLastName = inLastName;
    }
}
```

The person bean class of the sample program.

Web Application Deployment Descriptor

In order to be able to split the JSF configuration file into multiple files, the web application deployment descriptor has to be slightly modified. The welcome file list has been modified too, to make the introduction page the first one to be displayed.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <context-param>
        <param-name>com.sun.faces.verifyObjects</param-name>
        <param-value>false</param-value>
    </context-param>

    <context-param>
        <param-name>com.sun.faces.validateXml</param-name>
        <param-value>true</param-value>
    </context-param>

    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>server</param-value>
    </context-param>

    <context-param>
        <param-name>javax.faces.CONFIG_FILES</param-name>
        <param-value>/WEB-INF/faces-navigation.xml,/WEB-INF/faces-beans.xml</param-value>
    </context-param>

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

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

    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>

    <welcome-file-list>
        <welcome-file>faces/introduction.jspx</welcome-file>
    </welcome-file-list>
</web-app>
```

Web application deployment descriptor of the sample program.

JSF Configuration Files

Besides the faces-config.xml, two additional JSF configuration file exists for the sample application; faces-beans.xml and faces-navigation.xml. These additional files hold managed bean definitions and navigation configuration respectively.

faces-config.xml

With the managed beans and navigation in separate files, the main JSF configuration file is empty.

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

<!-- =========== MAIN CONFIGURATION FILE ================================== -->
<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd">

</faces-config>
```

The main JSF configuration file for the sample program.

faces-beans.xml

This configuration file contains all the managed beans of the sample program.

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

<!-- =========== MANAGED BEANS CONFIGURATION FILE ========================== -->
<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd">

    <managed-bean>
        <managed-bean-name>personBean</managed-bean-name>
        <managed-bean-class>com.ivan.model.PersonBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <managed-bean-name>introductionController</managed-bean-name>
        <managed-bean-class>com.ivan.controllers.IntroductionController</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <managed-bean-name>inputFirstNameController</managed-bean-name>
        <managed-bean-class>com.ivan.controllers.InputFirstNameController</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <managed-bean-name>inputLastNameController</managed-bean-name>
        <managed-bean-class>com.ivan.controllers.InputLastNameController</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <managed-bean-name>inputAddressController</managed-bean-name>
        <managed-bean-class>com.ivan.controllers.InputAddressController</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <managed-bean-name>completedController</managed-bean-name>
        <managed-bean-class>com.ivan.controllers.CompletedController</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <managed-bean-name>cancelledController</managed-bean-name>
        <managed-bean-class>com.ivan.controllers.CancelledController</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
</faces-config>
```

The configuration file for the JSF managed beans of the sample program.

faces-navigation.xml

This configuration file contains all the navigation configuration information of the sample program.

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

<!-- =========== NAVIGATION CONFIGURATION FILE ============================ -->
<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd">

    <navigation-rule>
        <from-view-id>/*</from-view-id>
        <navigation-case>
            <from-outcome>cancel</from-outcome>
            <to-view-id>/cancelled.jspx</to-view-id>
        </navigation-case>
    </navigation-rule>

    <navigation-rule>
        <from-view-id>/introduction.jspx</from-view-id>
        <navigation-case>
            <from-action>#{introductionController.doContinue}</from-action>
            <from-outcome>success</from-outcome>
            <to-view-id>/input_firstname.jspx</to-view-id>
            <redirect/>
        </navigation-case>
    </navigation-rule>

    <navigation-rule>
        <from-view-id>/cancelled.jspx</from-view-id>
        <navigation-case>
            <from-action>#{cancelledController.doRestart}</from-action>
            <from-outcome>success</from-outcome>
            <to-view-id>/input_firstname.jspx</to-view-id>
            <redirect/>
        </navigation-case>
    </navigation-rule>

    <navigation-rule>
        <from-view-id>/input_firstname.jspx</from-view-id>
        <navigation-case>
            <from-action>#{inputFirstNameController.doNext}</from-action>
            <from-outcome>success</from-outcome>
            <to-view-id>/input_lastname.jspx</to-view-id>
        </navigation-case>
    </navigation-rule>

    <navigation-rule>
        <from-view-id>/input_lastname.jspx</from-view-id>
        <navigation-case>
            <from-action>#{inputLastNameController.doNext}</from-action>
            <from-outcome>success</from-outcome>
            <to-view-id>/input_address.jspx</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-action>#{inputLastNameController.doPrevious}</from-action>
            <from-outcome>success</from-outcome>
            <to-view-id>/input_firstname.jspx</to-view-id>
        </navigation-case>
    </navigation-rule>

    <navigation-rule>
        <from-view-id>/input_address.jspx</from-view-id>
        <navigation-case>
            <from-action>#{inputAddressController.doNext}</from-action>
            <from-outcome>success</from-outcome>
            <to-view-id>/completed.jspx</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-action>#{inputAddressController.doPrevious}</from-action>
            <from-outcome>success</from-outcome>
            <to-view-id>/input_lastname.jspx</to-view-id>
        </navigation-case>
    </navigation-rule>

    <navigation-rule>
        <from-view-id>/completed.jspx</from-view-id>
        <navigation-case>
            <from-action>#{completedController.doRestart}</from-action>
            <from-outcome>success</from-outcome>
            <to-view-id>/input_firstname.jspx</to-view-id>
```

```
            <redirect/>
        </navigation-case>
        <navigation-case>
            <from-action>#{completedController.doPrevious}</from-action>
            <from-outcome>success</from-outcome>
            <to-view-id>/input_address.jspx</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

The navigation configuration file for the sample program.

Analysis

Navigation

The main point of this sample program is to study navigation in JSF. In this section some details of navigation configuration will be highlighted.
The navigation rule below is the rule that leads to the cancelled-page.

```
    <navigation-rule>
        <from-view-id>/*</from-view-id>
        <navigation-case>
            <from-outcome>cancel</from-outcome>
            <to-view-id>/cancelled.jspx</to-view-id>
        </navigation-case>
    </navigation-rule>
```

Navigation rule leading to the cancelled-page.

The <from-view-id> element, which could have been left out completely, specifies that the origin of the navigation can be any view in the application. When looking at the <navigation-case> below, the <to-view-id> element further specifies that it is possible to navigate to the cancelled view from any view. The <from-outcome> element puts a final restriction on the navigation to become: Navigation to the cancelled view will take place from any view in the application when the outcome of an action is “cancel”.

In the next navigation rule, the criteria has been narrowed further.

```
    <navigation-rule>
        <from-view-id>/introduction.jspx</from-view-id>
        <navigation-case>
            <from-action>#{introductionController.doContinue}</from-action>
            <from-outcome>success</from-outcome>
            <to-view-id>/input_firstname.jspx</to-view-id>
            <redirect/>
        </navigation-case>
    </navigation-rule>
```

Navigation rule leading from introduction view to input-first-name view.

The <from-view-id> element tells us that navigation can take place from the introduction view. The <navigation-case> contains a new element, the <from-action> element, which specifies exactly which action must be invoked, in order for the navigation to take place. The result is:
Navigation from the introduction view to the input-first-name view will take place if the action *#{introductionController.doContinue}* is invoked with the outcome “success”.
A quick peek in the introduction view controller class reveals the following method:

```
    public String doContinue()
    {
        return "success";
    }
```

The *doContinue* method from the introduction view controller class.

Thus we can see that the *#{introductionController.doContinue}* action will always succeed. The <from-outcome> element could be removed, but is left in place in the case that further outcomes and navigation cases are added in the future.
Finally, in the <navigation-case> element, there is a <redirect/> element. As mentioned in the Navigation section, this causes a redirect, rather than a forwarding, to the target of the navigation.

Further down in the navigation configuration file, there are navigation rules with more than one navigation case.

```
<navigation-rule>
    <from-view-id>/input_address.jspx</from-view-id>
    <navigation-case>
        <from-action>#{inputAddressController.doNext}</from-action>
        <from-outcome>success</from-outcome>
        <to-view-id>/completed.jspx</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-action>#{inputAddressController.doPrevious}</from-action>
        <from-outcome>success</from-outcome>
        <to-view-id>/input_lastname.jspx</to-view-id>
    </navigation-case>
</navigation-rule>
```

Navigation rule with two navigation cases.

The navigation rule can be read like this:
Navigation from the input-address view to the completed view will take place if the outcome of the *#{inputAddressController.doNext}* action is “success”. Navigation from the input-address view to the input-last-name view will take place if the outcome of the *#{inputAddressController.doPrevious}* action is “success”.

Splitting the JSF Configuration File

When a project grows larger, it can be very convenient to be able to split the JSF configuration file in multiple, smaller, files, as have been done in this sample program.
This is accomplished by adding the *javax.faces.CONFIG_FILES* context parameter to the web application deployment descriptor.

```
<context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-navigation.xml,/WEB-INF/faces-beans.xml</param-value>
</context-param>
```

Adding additional JSF configuration files in the web application deployment descriptor.

Note that, despite not being included in the list of configuration files, the faces-config.xml file will still be considered as a JSF configuration file and read by the JavaServer Faces implementation.

Giving Focus to Input Components

If the user is to input text, like in this sample program, it is very convenient if the first input field in which text is to be entered is given focus when the page loads.
The JavaScript function that takes a UI component identifier as a parameter and, if the component is found, gives focus to it has already been listed above.
Importing the script requires a little attention, since it is to be imported into a JSF (JSP) page that will be compiled into a servlet. Thus, a static import has to be used (include), which can be seen in the three input pages.

```
<script type="text/javascript">
    <jsp:directive.include file="focus.js"/>
</script>
```

Importing the JavaScript function into a JSF (JSP) page.

Next, the function needs to be invoked at an appropriate time. The *onload* attribute of the *body* tag comes in handy:

```
<body onload="focus('form:address_input')">
```

Invoking the focus function from a JSF web page.

Note also the component identifier passed to the focus function and make sure the components have been assigned identifiers.

Appendix 4 – Internationalization of a Web Page

As an example of the process of adding support for internationalization, we will look at a simple web page. This web page also enables the user to change the language of the web page, which in turn will showcase programmatic control of the current locale.

Web Page Before

The listing below shows the web page prior to adding support for internationalization. There will also be a controller, three message property files and a faces configuration file, but these will be presented later. The web page is to be stored in a file named “index.jspx”.

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

<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html">

    <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"/>
    <jsp:output omit-xml-declaration="no"
         doctype-root-element="html"
         doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
         doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

    <f:view>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title><h:outputText value="Internationalization"/></title>
            </head>
            <body>
                <h:form>
                    <p>
                        <h:outputText value="This web page shows how
internationalization can be easily accomplished in JavaServer Faces."/>
                    </p>
                    <p>
                        <h:outputText value="Choose a locale and click the Update
button."/><br/>
                        <h:outputFormat value="The locale has been changed {0} times.">
                            <f:param value="#{controller.changeCounter}"/>
                        </h:outputFormat>
                    </p>
                    <p>
                        <h:outputLabel for="locales" value="Language:"/>
                        <h:selectOneMenu id="locales"
                                         valueChangeListener="#{controller.newLocaleSele
cted}">
                            <f:selectItems value="#{controller.locales}"/>
                        </h:selectOneMenu>
                    </p>
                    <p>
                        <h:commandButton value="Update"/>
                    </p>
                </h:form>
            </body>
        </html>
    </f:view>
</jsp:root>
```

Web page index.jspx prior to adding support for internationalization.

Creating a Messages Property File

The first step is to create a property file and inserting the messages into the file, assigning a key for each message. The name of the first properties file is “messages.properties” and it is to be placed in an appropriate package along with the Java source code of the web application. In this example, a special package named “com.ivan.resources” for the messages is created.

```
title=Internationalization
presentation=This web page shows how internationalization can be easily accomplished in
JavaServer Faces.
instructions=Choose a locale and click the Update button.
changesCount=The locale has been changed {0} times.
localeLabel=Language:
updateButton=Update
```

The first, English, messages file “messages.properties” of the sample program.

The messages file just created is the default messages file, which language is English. Next, two additional message files are created; “messages_de.properties” and “messages_sv.properties” with the contents as below. Both files are to be located in the same package as the first messages file. These new message files contain German and Swedish messages, respectively.

```
title=Internationalisierung
presentation=Diese Webseite zeigt, wie Internationalisierung in den JavaServer Faces
leicht vollendet werden kann.
instructions=Wählen Sie eine Sprache und drucken Sie die Updatetaste an.
changesCount=Sprache ist {0} Zeiten geändert worden.
localeLabel=Sprache:
updateButton=Update
```

The German messages file “messages_de.properties” of the sample program.

```
title=Internationalisering
presentation=Denna websida visar hur internationalisering lätt kan åstadkommas i
JavaServer Faces.
instructions=Välj ett språk och tryck på Updatera knappen.
changesCount=Språket har ändrats {0} gånger.
localeLabel=Språk:
updateButton=Uppdatera
```

The Swedish messages file “messages_sv.properties” of the sample program.

Note the addition of “de” and “sv” to the file names. These are language codes which will be used later, when configuring the languages the web application supports.
The next step is to configure the faces configuration file so that the above messages can be used in the web page.

JSF Configuration File

The faces configuration file for this sample application contains configurations for:

- A resource bundle.
  This is the three messages files that were created in the previous section.
- Locale configuration.
  Configuration of which languages the web application supports without the need to modify or re-compile the source code.
- A managed bean.
  This is the controller for the web page of the sample program.

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

<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">

    <application>
        <resource-bundle>
            <base-name>com.ivan.resources.messages</base-name>
            <var>msgs</var>
        </resource-bundle>

        <locale-config>
            <default-locale>en</default-locale>
            <supported-locale>en</supported-locale>
            <supported-locale>de</supported-locale>
            <supported-locale>sv</supported-locale>
        </locale-config>
    </application>

    <managed-bean>
        <managed-bean-name>controller</managed-bean-name>
        <managed-bean-class>com.ivan.I18NController</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
</faces-config>
```

JSF configuration file faces-config.xml.

Web Application Deployment Descriptor

Only the welcome file list has been modified in the web application deployment descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
web-app_2_5.xsd">
    <context-param>
        <param-name>com.sun.faces.verifyObjects</param-name>
        <param-value>false</param-value>
    </context-param>

    <context-param>
        <param-name>com.sun.faces.validateXml</param-name>
        <param-value>true</param-value>
    </context-param>

    <context-param>
        <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
        <param-value>server</param-value>
    </context-param>

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

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

    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>

    <welcome-file-list>
        <welcome-file>faces/index.jspx</welcome-file>
    </welcome-file-list>
</web-app>
```

Web application deployment descriptor file, web.xml.

Controller

The controller is, as usual, a managed bean. First, the code is presented and then a short analysis follows. Note that this controller class is not “clean” MVC since it contains traces of presentation technology, such as the *ValueChangedEvent* and access to the *FacesContext* etc.
package com.ivan;

```
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Locale;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.faces.context.FacesContext;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ValueChangeEvent;

/**
 * Controller class for the internationalization application.
 *
 * @author Ivan A Krizsan
 */
public class I18NController
{
    /* Instance Variable(s): */
    private Locale mCurrentLocale;
    private Map<String, Locale> mLocales;
    private int mChangeCounter;

    /**
     * Initializes the managed bean.
     * Sets supported and default locale according to configuration
     * in the faces configuration file.
     */
    @PostConstruct
    public void initialize()
    {
        FacesContext theJSFContext = FacesContext.getCurrentInstance();
        Iterator<Locale> theLocaleIter;

        /* Retrieve supported locales and put them in a map the menu can use. */
        mLocales = new LinkedHashMap<String, Locale>();
        theLocaleIter = theJSFContext.getApplication().getSupportedLocales();

        while (theLocaleIter.hasNext())
        {
            Locale theLocale = theLocaleIter.next();
            mLocales.put(theLocale.getDisplayLanguage(), theLocale);
        }

        mCurrentLocale = theJSFContext.getApplication().
            getDefaultLocale();
    }

    /**
     * Processes value changed events for the locale selection menu.
     *
     * @param inEvent Event.
     */
    public void newLocaleSelected(ValueChangeEvent inEvent)
    {
        mCurrentLocale = (Locale)new Locale((String)inEvent.getNewValue());
        FacesContext.getCurrentInstance().getViewRoot().setLocale(mCurrentLocale);
        mChangeCounter++;
    }

    public Locale getCurrentLocale()
    {
        return mCurrentLocale;
    }

    public void setCurrentLocale(Locale inCurrentLocale)
    {
```

```
        mCurrentLocale = inCurrentLocale;
    }

    public Map<String, Locale> getLocales()
    {
        return mLocales;
    }

    public void setLocales(Map<String, Locale> inLocales)
    {
        mLocales = inLocales;
    }

    public int getChangeCounter()
    {
        return mChangeCounter;
    }

    public void setChangeCounter(int inChangeCounter)
    {
        mChangeCounter = inChangeCounter;
    }
}
```

Controller class for the JSF internationalization sample application.

Controller Analysis

The *initialize* method retrieves the supported locales and the default locale, as described in Retrieving Locale Configuration, and creates a map that holds the items selectable in the menu on the web page (<h:selectOneMenu> and <f:selectItems> tags in the web page). See the Multiple Items section in Making Selections.
The *newLocaleSelected* method is a value changed listener method, see Value Changed Events, which only will be called when the user selects a new locale. When a new locale is selected, the method will set the current locale of the current view to the selected locale, see Controlling the Locale.

Web Page After

Now all the messages in the original web page can be replaced with references to the resource bundle defined for the application. I have also added some code that outputs the default browser locale, found in the request object. The final result will look like this:

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

<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0"
          xmlns:f="http://java.sun.com/jsf/core"
          xmlns:h="http://java.sun.com/jsf/html"
          xmlns:c="http://java.sun.com/jsp/jstl/core">

    <jsp:directive.page contentType="text/html" pageEncoding="UTF-8"
        import="javax.servlet.jsp.jstl.core.Config"/>
    <jsp:output omit-xml-declaration="no"
         doctype-root-element="html"
         doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
         doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"/>

    <f:view>
        <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
                <title><h:outputText value="#{msgs.title}"/></title>
            </head>
            <body>
                <h:form>
                    <p>
                        <h:outputText value="#{msgs.presentation}"/>
                    </p>
                    <p>
                        <h:outputText value="#{msgs.instructions}"/><br/>
                        <h:outputFormat value="#{msgs.changesCount}">
                            <f:param value="#{controller.changeCounter}"/>
                        </h:outputFormat>
                    </p>
                    <p>
                        <c:set var="browserLocale" scope="request" value="$
{pageContext.request.locale}"/>
                        <h:outputFormat value="#{msgs.localeCodeLabel}">
                            <f:param value="#{requestScope.browserLocale}"/>
                        </h:outputFormat>
                    </p>
                    <p>
                        <h:outputLabel for="locales" value="#{msgs.localeLabel}"/>
                        <h:selectOneMenu id="locales"
                                         valueChangeListener="#{controller.newLocaleSele
cted}">
                            <f:selectItems value="#{controller.locales}"/>
                        </h:selectOneMenu>
                    </p>
                    <p>
                        <h:commandButton value="#{msgs.updateButton}"/>
                    </p>
                </h:form>
            </body>
        </html>
    </f:view>
</jsp:root>
```

Web page index.jspx after having added support for internationalization.

Appendix 5 - Building a WAR File

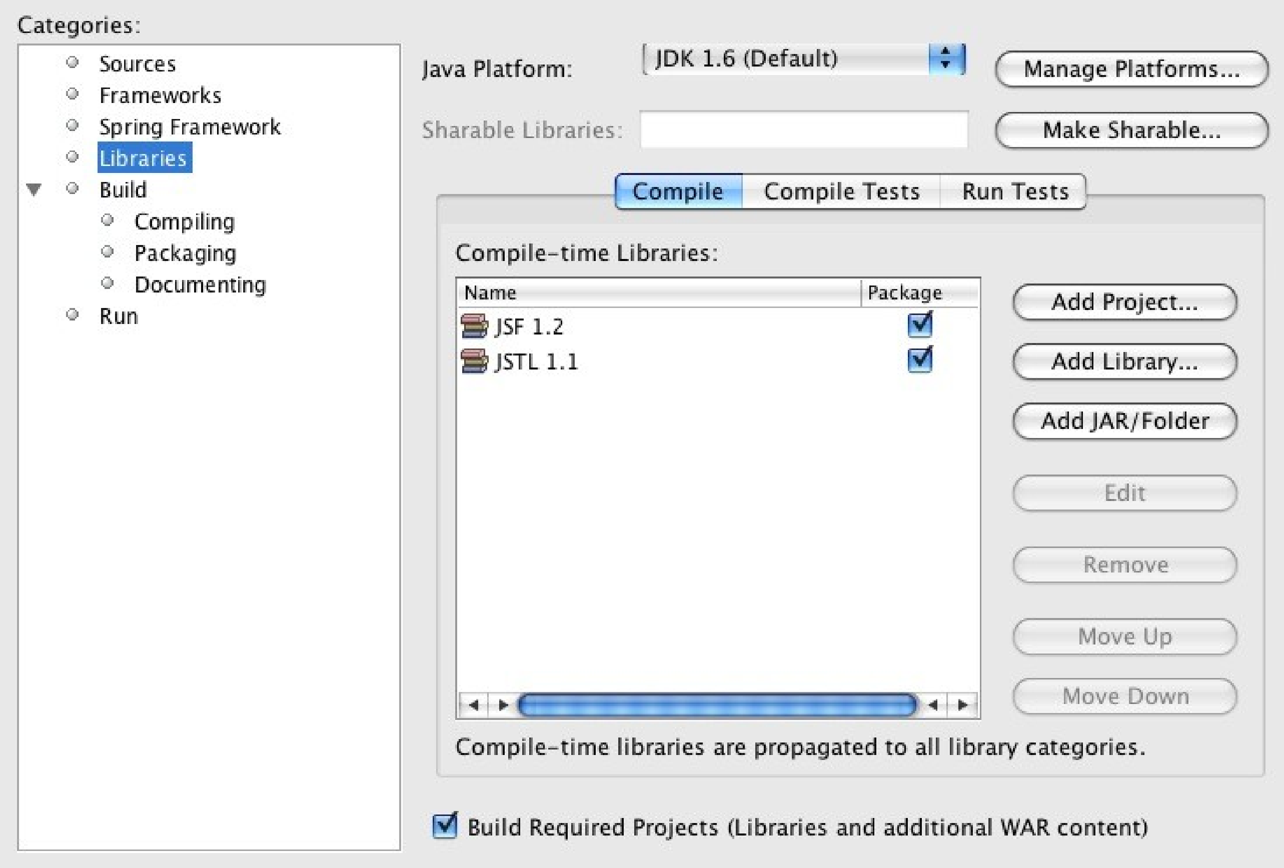
Having finished writing your latest JSF application, or any web application, you might want to pack it into a WAR file. WAR stands for web application archive, and is a file that contains the code and any additional resources of a web application, in order to make it easy and convenient to distribute.
The process of creating a WAR file differs between different development environments. Below, we'll look at how to accomplish this in NetBeans and Eclipse.
As example, you can use any of the web applications you have developed using JSF – I'll use the navigation sample program from appendix 3.
It is also assumed that the web container, Tomcat in my case, does not contain any JSF libraries needed by the web application.

Using NetBeans

In this section we'll look at how to create a WAR file in NetBeans and how to have some control over the contents of the WAR file.

Including Libraries

Right-click the project you want to build a WAR file for and select Properties in the menu that appears. In the window that appears, select the Libraries category. The window should now look like this:

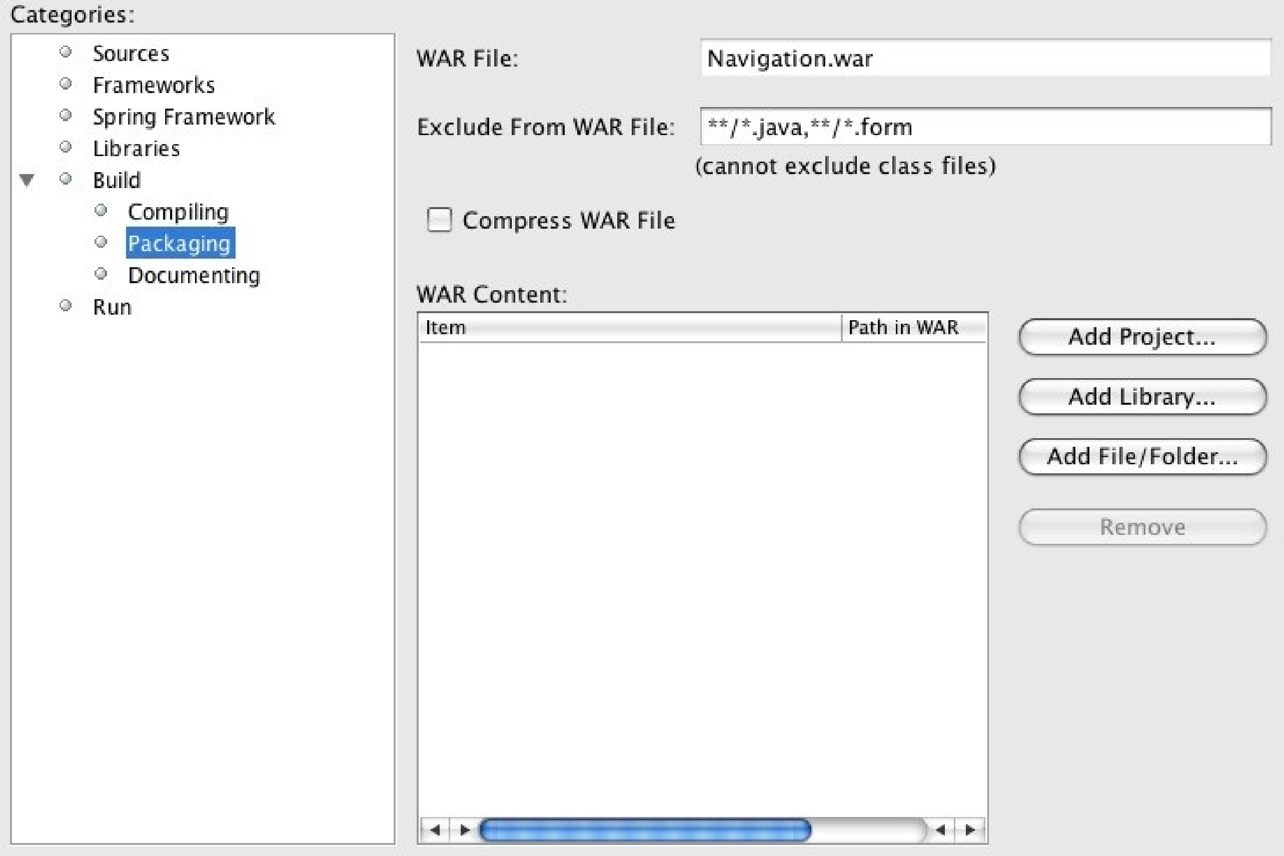


Managing the libraries for a JSF web application in NetBeans.

The project uses the JSF 1.2 and JSTL 1.1 library.
The Package check-box to the right of the library name indicates whether the library should be included in the WAR file or not. If you have multiple web applications that uses the same libraries, you might want to put the libraries in the web container and un-check these check-boxes, instead of including them in the web applications. How to do this depends on the web container used, and is out of the scope of this document.
At the bottom of the window, there is another check-box saying “Build Required Projects (Libraries and additional WAR content)”. Leave this check-box checked, since you probably will want the latest version of required projects etc. when building your web application.

Including Additional Content

Next, select the Packaging category that is a sub-category of the Build category.



Managing WAR file packaging for a web application in NetBeans.

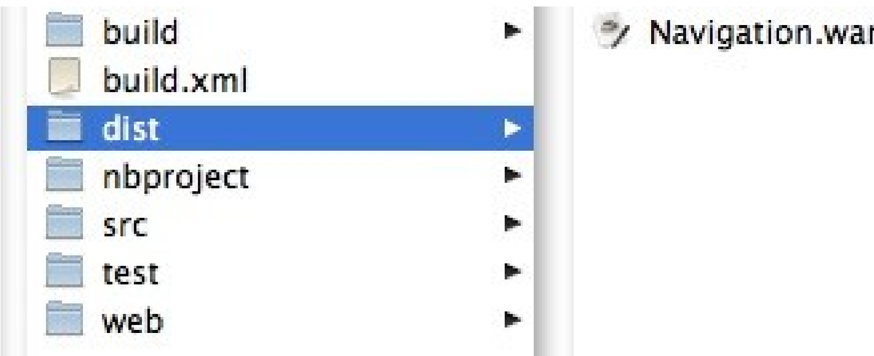
Here you can configure the following:

- The name of the WAR file that your web application will be stored in.
- Files that are not to be included in the WAR file.
  Source-code files should not be included in the WAR file, if there isn't a special reason for doing so etc.
- Whether or not the WAR file should be compressed.
- Additional content of the WAR file.
  The libraries that are to be packed will automatically be included in the WAR file. Here you have a chance to add additional projects, libraries, files etc. that you want included in the WAR file.

When finished, click the OK button to save the configuration.

Building the WAR File

Finally, in order to build the WAR file, right-click the project and select “Build” or “Clean and Build” in the menu that appears. The WAR file can be found in the “dist” directory in the project directory.



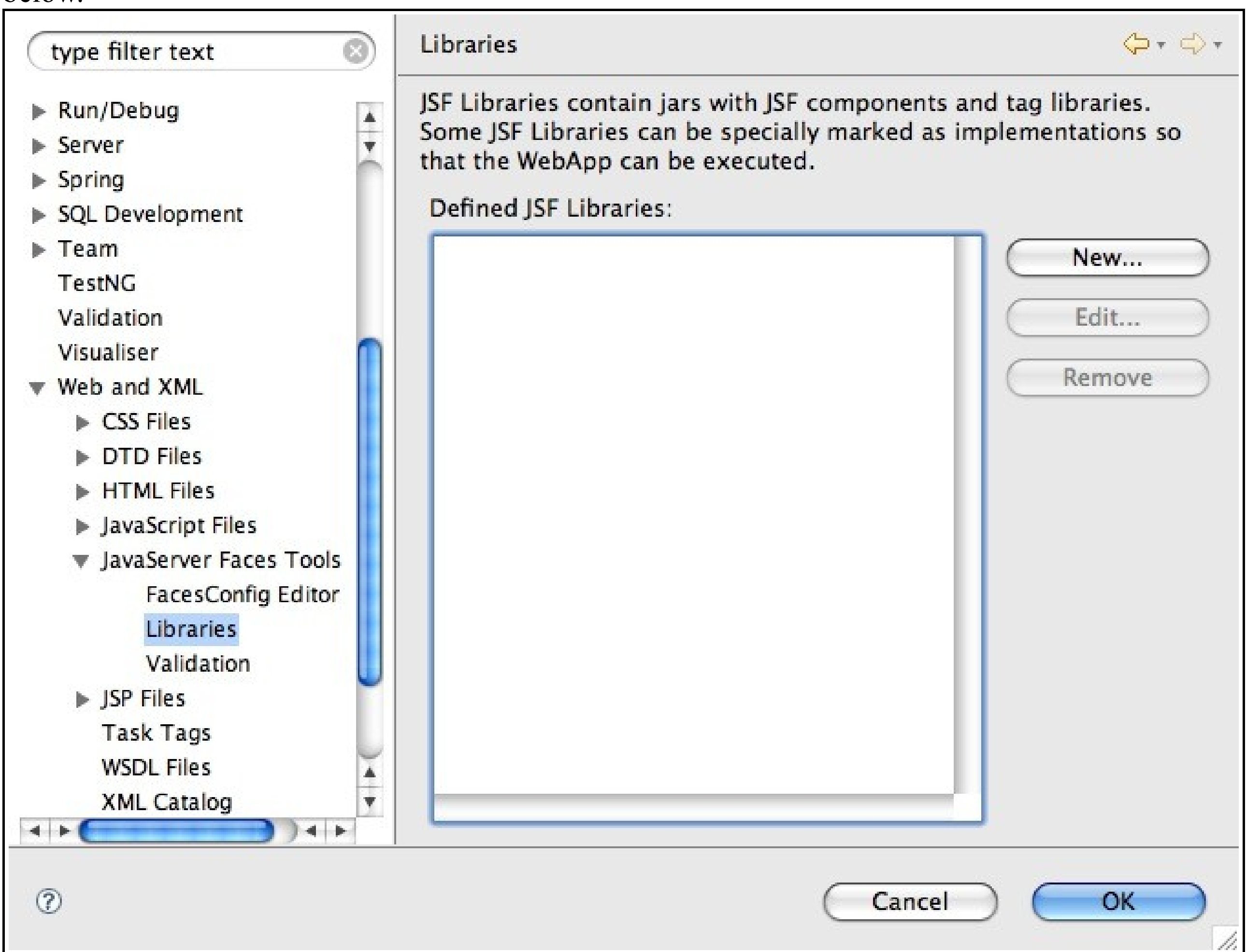
The WAR file built by NetBeans is located in the “dist” directory in the project directory.

Using Eclipse

It is assumed that Eclipse with the web tools platform, including JSF support, is already installed. If not, it can be downloaded from http://www.eclipse.org.
Using Eclipse to develop JSF web applications requires one step of preparation, namely registration of the JSF libraries.

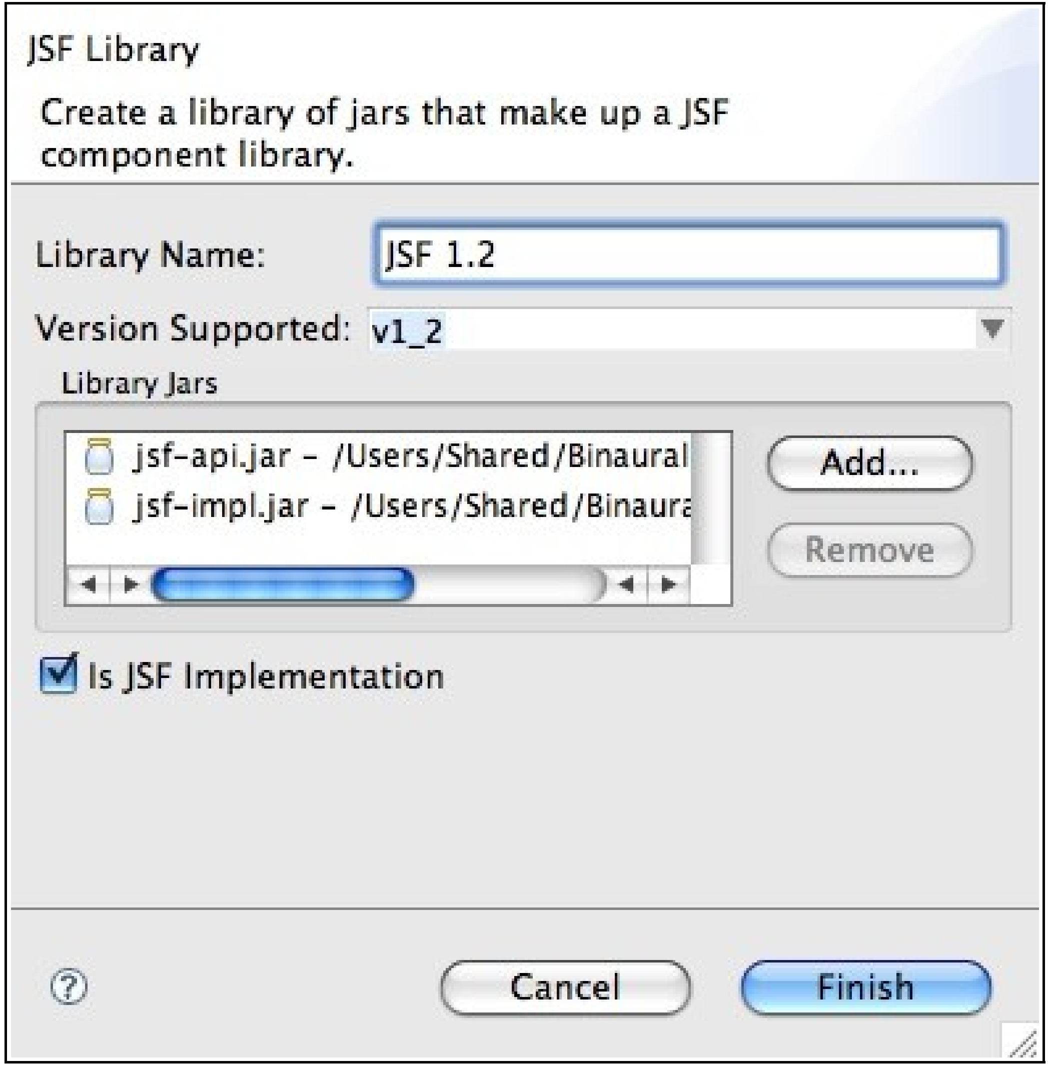
Register JSF Libraries

Make sure you have the two JSF library files "jsf-api.jar" and "jsf-impl.jar" for the latest version of JSF. These can be downloaded from http://java.sun.com/javaee/javaserverfaces/download.html as part of the binary distribution package.
Open the Eclipse preferences and go to the Libraries tab under JavaServer Faces Tools, see picture below.



Registering JSF libraries in the Eclipse preferences – step one.

Click the New... button on the right.



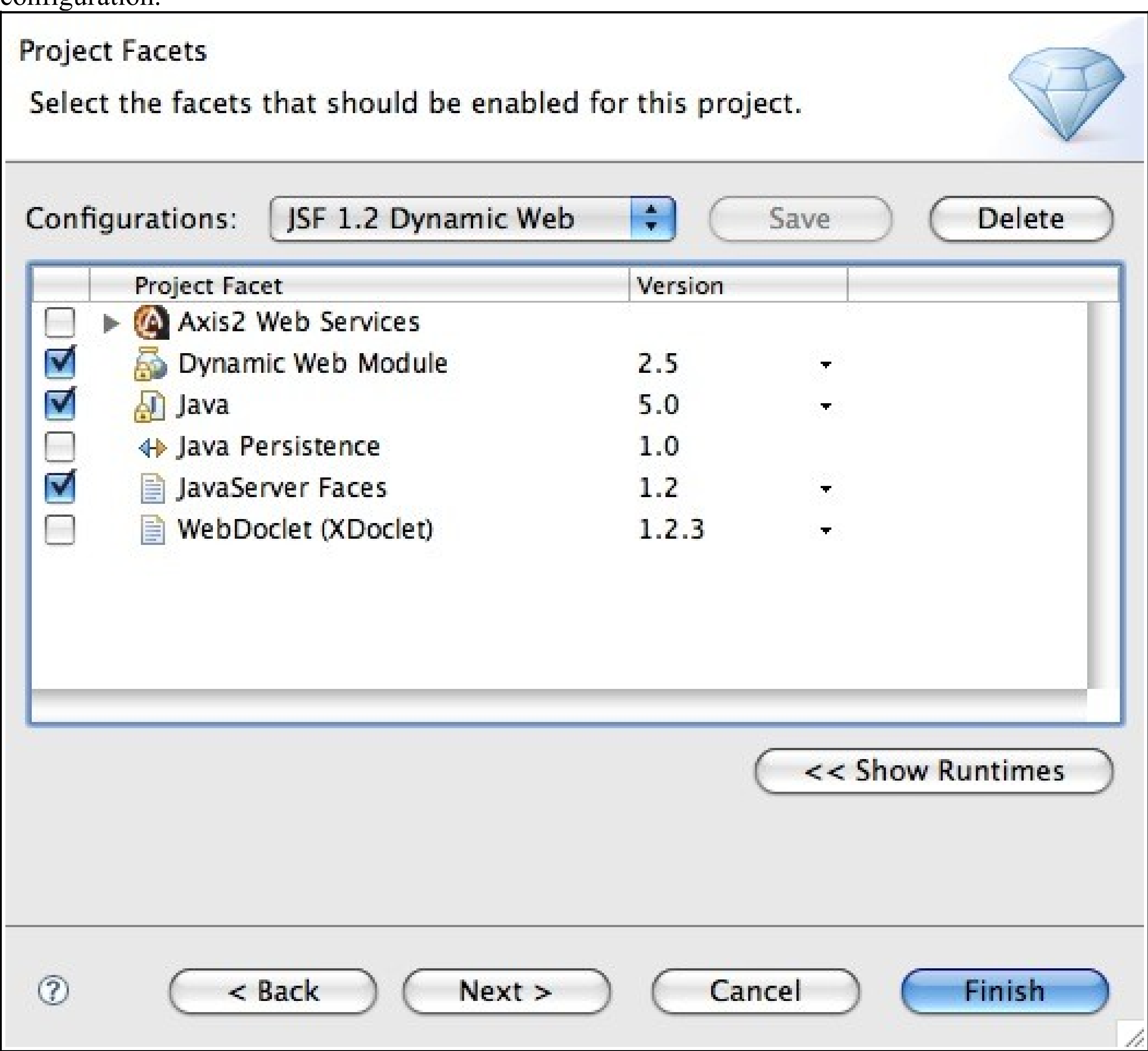
Registering JSF libraries in Eclipse – step two.

- Enter a name for the library, “JSF 1.2” in my case.
- Add the two JSF library files downloaded earlier.
- Check the “Is JSF Implementation” check-box, if it is not already checked.
- Click the Finish button.

You are now ready to create a JSF web application project.

Creating the Project

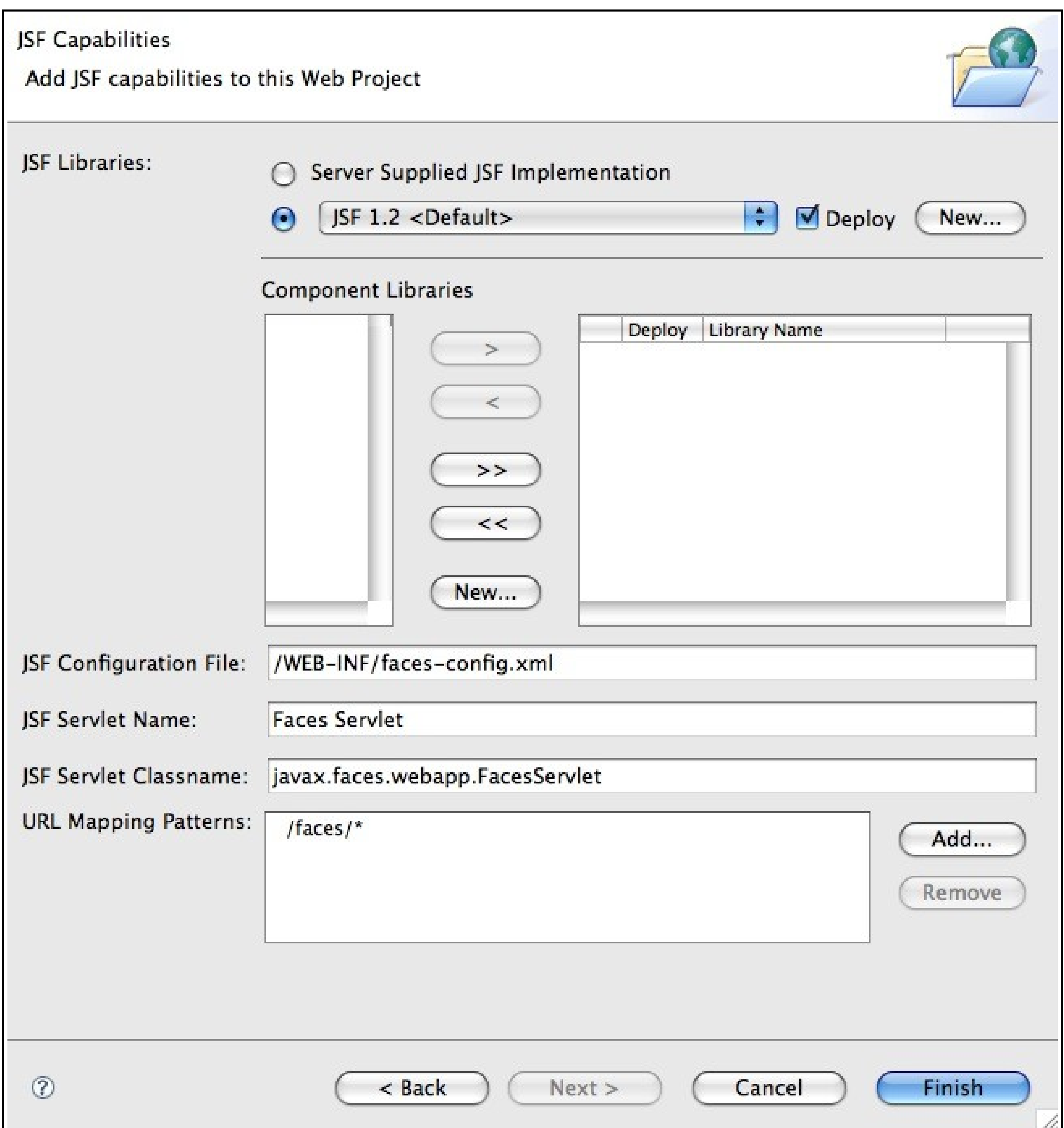
When creating the dynamic web application project in which your JSF web application will be developed, you need to include the JSF libraries registered earlier.
When creating the new project and specifying project facets, select the JSF 1.2 Dynamic Web configuration.



Project facets configuration for a dynamic web project that uses JSF in Eclipse.

When later being asked to add JSF capabilities, you have two choices:

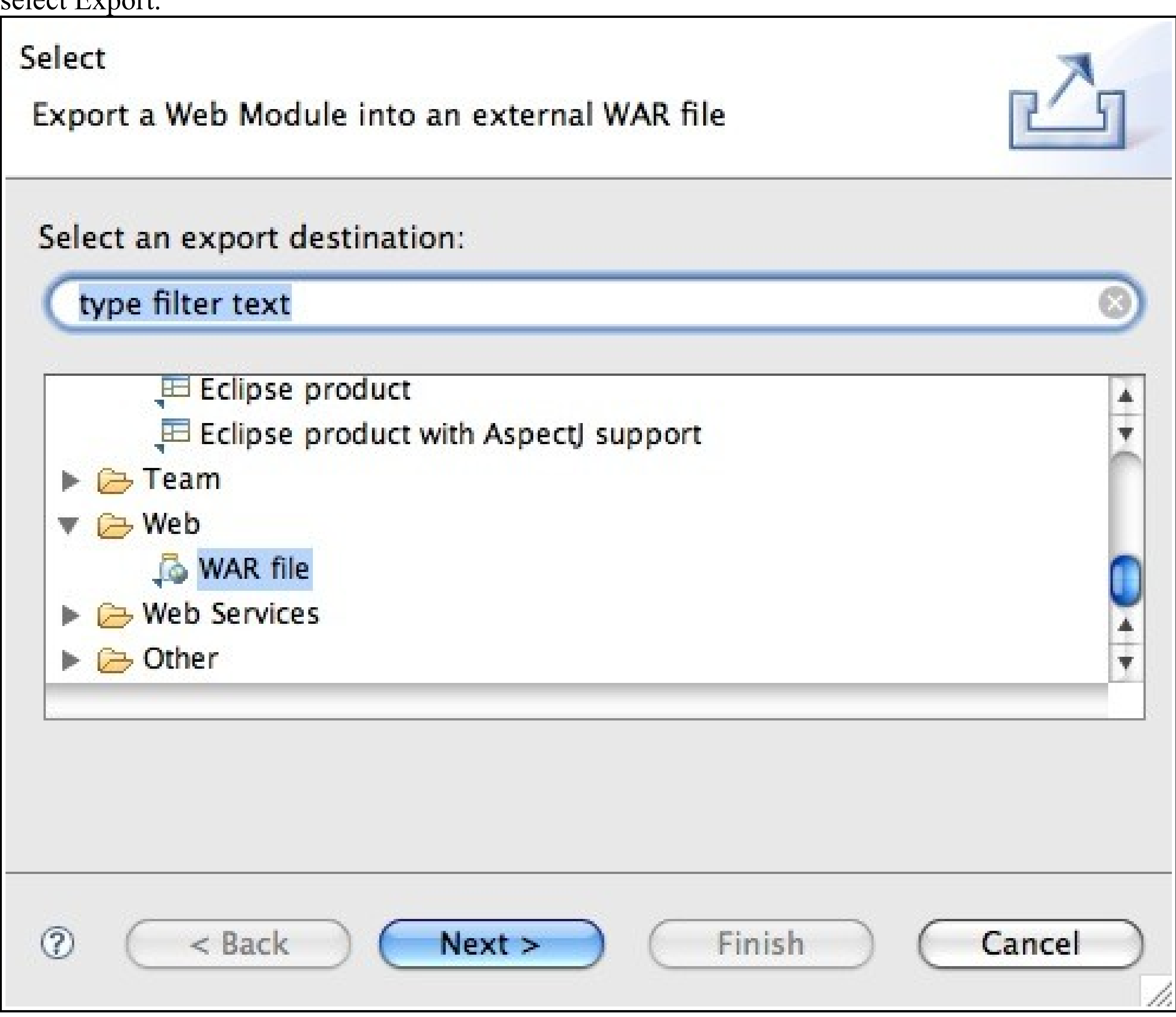
- Server Supplied JSF Implementation
  Select this if your web container already includes JSF libraries.
- Custom configured JSF libraries.
  These are the libraries we prepared in the previous step, which we will use now.
  Also make sure that the Deploy check-box is checked, in order to deploy the libraries with the web application, assuming that the web container does not have the JSF libraries.



Configuring JSF capabilities for the new dynamic web project in Eclipse.

Building the WAR File

Finally, after having developed the JSF web application, we are read to build the WAR file. Select the web application project in the Project or Package Explorer, then go to to File menu and select Export.



Exporting a (JSF) web application project to a WAR file in Eclipse.

After having clicked the Next> button, you will be asked where to save the WAR file. Click the Finish button to export the WAR file to the chosen location.