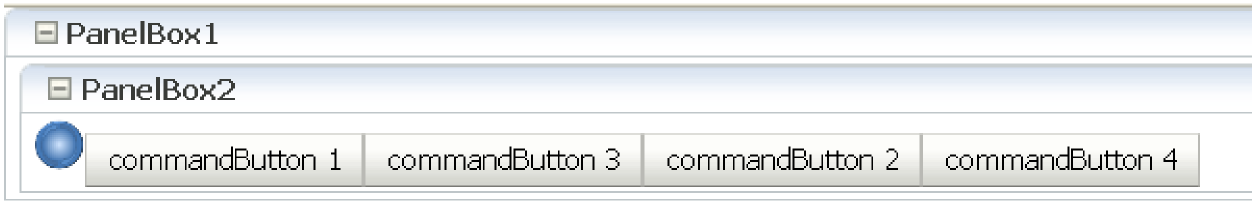
Creating a Custom JSF 1.2 Component - With Facets, Resource Handling, Events and Listeners

This article demonstrates the quick step approach to creating a new custom component in the old fashioned way (that means: it is not a Facelets template based or an ADF Faces 11g Declarative Component). Its primary purpose is to help me quickly retrace my steps. But perhaps it will benefit some of you as well.

The Shuffler component I will develop supports facets. It will render its facet children - one after the other. Which one is rendered first can be indicated through an attribute facetOrder (values normal, reverse and random), which is EL enabled. A shuffler-method-expression can optionally be set to provide the Shuffler with a shuffle-order-processor: the method is invoked with the list of facets to shuffle and will return it in the order in which to render the children.

The component can render with a shuffle icon that when pressed causes the children to be shuffled. The Shuffler component allows registration of Shuffle Event Listeners, custom listeners that are informed whenever the shuffle event occurs.

An example of how the Shuffler can be used inside a JSF page:



Some elements of custom JSF components that are explicitly discussed in this article:

- dynamic attributes of type ValueExpression (EL enabled)
- attributes of type MethodExpression (also EL enabled)
- facets
- (custom) events and listeners

Bare essentials for custom JSF components

A custom JSF component is represented by a Java Class - one that extends from UIComponentBase. An instance of this class is created whenever a new page is rendered that contains the component (and for each occurrence of the component in the page, a new instance of the class is created). The component class holds the attributes that are set by the page developer and that determine the behavior and appearance of the component. The component class has the internal logic of the component and it deals for example with events and listeners. This class may also render the markup (HTML) for the component - though it is a better practice to leave the actual rendering to a Renderer class.

```
public class Shuffler extends UIComponentBase {
...
}
```

Most custom JSF component will also have an associated Renderer class, that extends from Renderer. Note that some components will not actually be rendered (such as Listeners, Iterators or Parameters) and therefore will not have a Renderer class. The Renderer is not only responsible for rendering the HTML, it will also inspect (decode) the incoming request from the browser to see whether the request parameter map contains values that are of interest to the component - that indicate for example that a value has been entered or set on the component('s representation in the browser) or an action has been executed against it. Note that one JSF component may have multiple Renderers, for example for different channels and protocols (to render a representation of the component in plain

XML, in WML, in JavaFX or XUL) or for different user agents (Firefox, Internet Explorer) or themes (professional user, internet surfer).

```
public class ShufflerRenderer extends SuperRenderer {
...
}
```

JavaServer Faces pages can be created in various ways - including programmatically, using Facelets and using JSP pages. The latter option, through JSP, is still the most common one, though that is about to change with JSF 2.0 favoring Facelets. Page developers using JSPs will describe the JSF component tree that will need to be instantiated in memory for rendering a certain View using a plain JSP page. The tags in the JSP page are normal JSP tags - described by TLD (tag library descriptors) - corresponding to JSF components and therefore JSF component classes. Every JSF component that needs to be used in JSPs has to have a corresponding JSP tag-class, one that will typically extend from UIComponentELTag (or just from TagSupport when no JSF component is added to the component tree for a certain tag, for example when that tag represents a listener or parameter). The Tag Class specifies which JSF Component it represents. It also indicates which Renderer should be used to render the component. This means that one component can have multiple JSP tags associated with it, each providing a different way of rendering the component. Note: the renderer can also be specified dynamically - taking user preferences or characteristics into account

```
public class ShufflerTag extends UIComponentELTag {

public static final String COMPONENT_TYPE = "nl.amis.jsf.UIShuffler";
public static final String RENDERER_TYPE  = "nl.amis.jsf.ShufflerRenderer";
public String getComponentType() {
return COMPONENT_TYPE;
}

public String getRendererType() {
return RENDERER_TYPE;
}
...
}
```

Tags representing JSF components need to be described in TLD files (Tag Library Descriptors) just like any other JSP tag.The entry in the TLD defines the tag label to use in the page, whether the tag can contain child-tags, some descriptive meta data and every attribute that can be configured in the tag. For each attribute the TLD-entry specifies the type, whether it is required and if the attribute can contain an EL expression passing in a value or an EL expression passing in a method; in the latter case, the entry also prescribes the signature of the method:

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<taglib xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaeehttp://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd"
version="2.1" xmlns="http://java.sun.com/xml/ns/javaee">
<display-name>ShufflerLib</display-name>
<tlib-version>1.0</tlib-version>
<short-name>ShufflerLib</short-name>
<uri>/nl.amis,jsf/ShufflerLib</uri>
<tag>
```

```
<description>Writes a DIV element that contains the facets in a specific order.</description>
<name>shuffler</name>
<tag-class>nl.amis.jsf.shuffler.ShufflerTag</tag-class>
<body-content>JSP</body-content>
<!-- standard UIComponent attributes -->
<attribute>
<name>id</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>rendered</name>
<required>false</required>
<deferred-value>
<type>boolean</type>
</deferred-value>
</attribute>
<attribute>
<name>binding</name>
<required>false</required>
<deferred-value>
<type>javax.faces.component.UIComponent</type>
</deferred-value>
</attribute>
<!-- custom attributes -->
<attribute>
<name>styleClass</name>
<required>false</required>
<deferred-value>
<type>java.lang.String</type>
</deferred-value>
</attribute>
.....
</tag>
```

JSF components need to be registered in a special faces-config.xml file (special in the sense that it is not the faces-config.xml that drives a web application but rather one that acts like a repository of components and their renderers. Note however that all entries in this special faces-config.xml is merged together with the ‘normal’ faces-config.xml. That means in turn that while the special file is primarily seen as the registry of components, it can also configure PhaseListeners, Navigation Rules (hard to see the value in that) and Managed Beans (which can be very useful).

The component registration in faces-config.xml consists of a component type that is associated with a the component class.

```
<component>
<component-type>nl.amis.jsf.UIShuffler</component-type>
<component-class>nl.amis.jsf.shuffler.Shuffler</component-class>
</component>
```

Renderers can also be registered in this file. A renderer entry registers a renderer-type (corresponding to the value returned by the getRendererType() method in the tag class) associated with the RendererClass. Based on the value (rendererType) returned by the tag class, the correct class to instantiate can be determined from this entry:

```
<render-kit>
<renderer>
<component-family>nl.amis.Shuffler</component-family>
<renderer-type>nl.amis.jsf.ShufflerRenderer</renderer-type>
<renderer-class>nl.amis.jsf.shuffler.ShufflerRenderer</renderer-class>
</renderer>
</render-kit>
```

Implementing the Classes: Component, Renderer and TagHandler

The TagHandler ShufflerTag is the intermediary between the world of JSP pages (and the Servlet/JSP engine that translates the JSP file into a servlet class) and the JSF realm. Every tag in the JSP page needs to be turned into its corresponding JSF representation. The tag handler needs to override the setProperties() method inherited from the UIComponentELTag class; this method takes all the values set on the tag attributes in the page and passes them onwards to the Component. In our initial case, the tag is used in JSPs like this:

```
<shf:shuffler styleClass="h1" id="s1" rendered="true">
... other content
</shf:shuffler>
```

The styleClass attribute is the only one we defined - id and rendered are defined on every JSP-tag based on JSF's UIComponentELTag. Thye styleClass attribute is also the only attribute we need to take responsibility for in the tag class, by providing a setter method that sets a private member and by passing the value of that private member to the component in the setProperties() method. The code for the ShufflerTag class now becomes:

```
package nl.amis.jsf.shuffler;

import javax.el.ValueExpression;

import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentELTag;

public class ShufflerTag extends UIComponentELTag {

public static final String COMPONENT_TYPE = "nl.amis.jsf.UIShuffler";
public static final String RENDERER_TYPE = "nl.amis.jsf.ShufflerRenderer";

private ValueExpression styleClass;

public String getComponentType() {
return COMPONENT_TYPE;
}

public String getRendererType() {
return RENDERER_TYPE;
}
```

```java

protected void setProperties(UIComponent component) {
super.setProperties(component);
processProperty(component, styleClass, Shuffler.STYLECLASS_ATTRIBUTE_KEY);
}

public void release() {
super.release();
styleClass= null;
}

protected final void processProperty(final UIComponent component, finalValueExpression property,
final String propertyName) {
if (property != null) {
if(property.isLiteralText()) {
component.getAttributes().put(propertyName, property.getExpressionString());
}
else {
component.setValueExpression(propertyName, property);
}
}
}

public void setStyleClass(ValueExpression styleClass) {
this.styleClass = styleClass;
}
}
```

We cater for the fact that styleClass can contain a ValueExpression - as all attributes can, starting from JSF 1.2. In the method processProperty we check whether the string passed for styleClass is a literal string or should be considered an EL expression. In the latter case, we pass a ValueExpression to the component, otherwise a ‘normal’ attribute. Also note that the super class takes care of the attributes id, rendered and binding. However, we do have to specify them in the tag-library.

The component class in our case leads a pretty comfortable life: the tag handler informs him of all the attribute values and the actual rendering is left to a special Renderer class. The component is a pretty passive element in this simple example:

```java
package nl.amis.jsf.shuffler;

import javax.faces.component.UIComponentBase;
import javax.faces.context.FacesContext;

public class Shuffler extends UIComponentBase {

public static final String FAMILY = "nl.amis.Shuffler";
public static final String STYLECLASS_ATTRIBUTE_KEY = "styleClass";

public String getFamily() {
return FAMILY;
}

@Override
public Object saveState(FacesContext facesContext) {
Object values[] = new Object[2];
values[0] = super.saveState(facesContext);
values[1] = this.getAttributes().get(STYLECLASS_ATTRIBUTE_KEY);
return values;

}

@Override
public void restoreState(FacesContext facesContext, Object state) {
Object values[] = (Object[])state;
super.restoreState(facesContext, values[0]);
this.getAttributes().put(STYLECLASS_ATTRIBUTE_KEY, values[1]);
}
}
```

The only really useful thing the component does is implementing the saveState and restoreState methods. These methods play an important part in turning the state of the component into a serializable object array and restoring that state of the component in the RestoreView phase, based on the serialized array.

The Tag Handler specifies in its getRendererType() method that the renderer to use for this component when using the shuffler tag, is one called nl.amis.jsf.ShufflerRenderer. In the faces-config.xml file, we have indicated that this renderer type is associated with the class nl.amis.jsf.shuffler.ShufflerRenderer that extends Renderer. The renderers in JSF can override methods like encodeBegin(), encodeEnd(), encodeChildren() and decode() - the latter only when we have to process the incoming request, looking for new values set on or events that occurred on the component. In our case, we initially will simply have the ShufflerRenderer render a DIV element with a class attribute (based on the styleClass attribute). The DIV will allow the children of the Shuffler component to render - by not overriding the encodeChildren() method.

```
package nl.amis.jsf.shuffler;

import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.render.Renderer;

public class ShufflerRenderer extends Renderer {

@Override
public void encodeBegin(final FacesContext facesContext,
final UIComponent component) throws IOException {
super.encodeBegin(facesContext, component);
final ResponseWriter writer = facesContext.getResponseWriter();

writer.startElement("DIV", component);
String styleClass =
(String)attributes.get(Shuffler.STYLECLASS_ATTRIBUTE_KEY);
writer.writeAttribute("class", styleClass, null);
}

@Override
public void encodeEnd(final FacesContext facesContext,
final UIComponent component) throws IOException {
final ResponseWriter writer = facesContext.getResponseWriter();
writer.endElement("DIV");
}
}
```

Next steps - working with facets

The Shuffler component is created to dynamically (re)order its child contents. It will do so using facets. The content you want this component to shuffle is passed in two or more facets. The facets are named using string representations of integers, so for example:

```
<shf:shuffler styleClass="mySpecialStyle"facetOrder="reverse" id="s1" >
<f:facet name="1">
... content
</f:facet>
<f:facet name="2">
... content
</f:facet>
<f:facet name="3">
... content
</f:facet>
</shf:shuffler>
```

Facets are automatically supported on JSF components. The getFacets() method is available inside the Shuffler component class and will return a collection of facet UIComponents. Facets are special children for a JSF component: the framework will never render the contents of facets on its own. It is up to the component to determine when and how to render the contents of its facets. So, there is some work to do for the ShuffleRenderer class. But first we need to add support for the new facetOrder attribute. Adding an attribute means:

- adding an attribute entry in the TLD
- adding support for processing the attribute in the Tag Handler (a setter and a line of code in setProperties())
- adding the attribute in saveState() and restoreState() in the Component class
- Here we go:

In the tld entry, add:

```
<attribute>
<name>facetOrder</name>
<required>false</required>
<deferred-value>
<type>java.lang.String</type>
</deferred-value>
</attribute>
```

In the tag-handler class ShufflerTag add:

```
private ValueExpression facetOrder;

```

```
public void setFacetOrder(ValueExpression facetOrder) {
this.facetOrder = facetOrder;
}
```

and in setProperties():

```
processProperty(component, facetOrder, Shuffler.FACETORDER_ATTRIBUTE_KEY);
```

Finally in the component class Shuffler , add:

```
public static final String FACETORDER_ATTRIBUTE_KEY = "facetOrder";
@Override
public Object saveState(FacesContext facesContext) {
Object values[] = new Object[3];
values[0] = super.saveState(facesContext);
values[1] = this.getAttributes().get(STYLECLASS_ATTRIBUTE_KEY);
values[2] = this.getAttributes().get(FACETORDER_ATTRIBUTE_KEY);
return values;
}

@Override
public void restoreState(FacesContext facesContext, Object state) {
Object values[] = (Object[])state;
super.restoreState(facesContext, values[0]);
this.getAttributes().put(STYLECLASS_ATTRIBUTE_KEY, values[1]);
this.getAttributes().put(FACETORDER_ATTRIBUTE_KEY, values[2]);
}
```

The Shuffler also needs to make the facets available to the renderer, in the order that is prescribed by the facetOrder attribute. This attribute supports three values: normal, reverse and random.

```
public List<UIComponent> getOrderedFacets(FacesContext facesContext) {
// allowable values:  normal (default) and reverse
// the normal order of the facets is determined by ordering the facets by name (assuming the facetnames are string representations of integers)

// create a sorted list with the integers representing the facets
List<Integer> facetIndexValues = new ArrayList();
List<String> facetNames = new ArrayList(getFacets().keySet());
for (String facetName : facetNames) {
facetIndexValues.add(new Integer(facetName));
}
Collections.sort(facetIndexValues);

// create the list of facets corrresponding to the sorted list of facet index values
List<UIComponent> orderedFacets = new ArrayList();
for (Integer index : facetIndexValues) {
orderedFacets.add(getFacets().get(index.toString()));
}

// depending on the value for the facetOrder attribute, we may need to reorganize the orderedFacets list
String facetOrder =
(String)this.getAttributes().get(Shuffler.FACETORDER_ATTRIBUTE_KEY);
if ("reverse".equalsIgnoreCase(facetOrder)) {
```

```
Collections.reverse(orderedFacets);
} else if ("random".equalsIgnoreCase(facetOrder)) {
Collections.shuffle(orderedFacets);
} else if ("normal".equalsIgnoreCase(facetOrder)) {
// need to do nothing as with normal the order returned by getFacets() is the correct one

}
return orderedFacets;
}
```

The ShufflerRenderer will have to do the real work. It will retrieve the facets - in the proper order - from the Shuffler Component class and ask JSF to render them.

```
package nl.amis.jsf.shuffler;

import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.render.Renderer;
import javax.faces.component.UIComponent;

public class ShufflerRenderer extends Renderer {

@Override
public void encodeBegin(final FacesContext facesContext,
final UIComponent component) throws IOException {
super.encodeBegin(facesContext, component);
final ResponseWriter writer = facesContext.getResponseWriter();

writer.startElement("DIV", component);
String styleClass =
(String)attributes.get(Shuffler.STYLECLASS_ATTRIBUTE_KEY);
writer.writeAttribute("class", styleClass, null);

List<UIComponent> orderedFacets = ((Shuffler)component).getOrderedFacets(facesContext);
for (UIComponent facet:orderedFacets) {
facet.encodeAll(facesContext);
}
}

@Override
public void encodeEnd(final FacesContext facesContext,
final UIComponent component) throws IOException {
final ResponseWriter writer = facesContext.getResponseWriter();
writer.endElement("DIV");
}
}
```

With these changes, we can now add real content to the Shuffler and have it rendered, in the order we specified - which can be random. Also note that we can use an EL expression to have the facetOrder dynamically derived:

```
<shf:shuffler styleClass="mySpecialStyle" <b>facetOrder="#{bean.liveFacetOrder}"</b> id="s1" >
<f:facet name="1">
... content
</f:facet>
<f:facet name="2">
... content
</f:facet>
<f:facet name="3">
... content
</f:facet>
</shf:shuffler>
```

Downloading Resources

The next step in our exploration of the development of custom JSF components is the addition of resources like images and JavaScript libraries. Note that in JavaServer Faces 2.0 a new facility is available, especially for this purpose. However, in our 1.2 setting we have to come up with something ourselves. That is not to say no solutions exist for JSF 1.2; almost every library comes with a form of resource handling. Then there is the Weblet framework that was introduced especially for this purpose. Another option leverages JSF itself: its capability through PhaseListeners to intercept a request, interpret the requested ViewId and optionally serve up an image or JS file in response to the request. This approach is proposed in JavaServer Faces, The Complete Reference by Ed Burns and Chris Schalk. I have slightly modified there code for my own purposes. However, the central idea clearly is theirs.

My objective is to add an image to the Shuffler component. The next step will be to allow the user to click on the image and by doing so tgrigger a re-shuffle. But that part is for later, first add the image itself.

The HTML rendered by the ShufflerRenderer needs to be extended with the IMG tag, that is easy enough. Less trivial is the value for the SRC attribute on the IMG tag.

The change in the encodeBegin method in the ShufflerRenderer:

```
writer.startElement("IMG", component);
writer.writeAttribute("src", imageUrl( facesContext,SHUFFLE_IMAGE), null);
writer.writeAttribute("alt", "Click to reshuffle", null);
writer.writeAttribute("width", "20px", null);
writer.endElement("IMG");
```

With SHUFFLE_IMAGE specified as:

```
private static String SHUFFLE_IMAGE = "shuffleIcon.png";
```

The imageUrl() method is defined as follows

```
private final static String IMAGE_PATH ="/faces/images/";

protected String imageUrl(FacesContext facesContext, String image) {
ViewHandler handler = facesContext.getApplication().getViewHandler();
String imageUrl =
handler.getResourceURL(facesContext, IMAGE_PATH + image);
return imageUrl;
}
```

The URLs for images are now constructed to look like this:

http://somehost:7101/CustomJSFConsumer/faces/images/shuffleIcon.png

The request for the shuffleIcon.png that is sent by the browser should be intercepted by a component that knows how to handle it. Because of the /faces/ part, this request is sent to the FacesServlet and processed through the JSF lifecycle. The componoent to intercept it will be a phaseListener that fires after restore view. It inspects the ViewId. When the ViewId contains the predefined indicator ("/images/") it steps in and takes over processing of the request. It will find the name of the image that is requested by taking the part of the ViewId that comes after /images/. It will then locate the image file on the classpath (that works well for a component packaged in a jar file, it can have the images packaged in the jar file too), looking for a directory called /images/ - as specified by the IMAGE_PATH constant. It copies the image from the file to the outputstream after setting the content type.

```
package nl.amis.jsf;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

import java.net.URL;
import java.net.URLConnection;

import javax.faces.context.FacesContext;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletResponse;

public class ResourceServerPhaseListener implements PhaseListener {
public ResourceServerPhaseListener() {
super();
}

public PhaseId getPhaseId() {
return PhaseId.RESTORE_VIEW;
}

public void afterPhase(PhaseEvent event) {
// If this is restoreView phase
if (PhaseId.RESTORE_VIEW == event.getPhaseId()) {
if (-1 !=
event.getFacesContext().getViewRoot().getViewId().indexOf(RENDER_IMAGE_TAG)) {
// extract the name of the image resource from the ViewId
String image =
event.getFacesContext().getViewRoot().getViewId().substring(event.getFacesContext()
.getViewRoot().getViewId().indexOf(RENDER_IMAGE_TAG) +
RENDER_IMAGE_TAG.length());
// render the script
writeImage(event, image);
event.getFacesContext().responseComplete();
}
}
}

public void beforePhase(PhaseEvent event) {
}

public static final String RENDER_IMAGE_TAG = "/images/";
public static final String IMAGE_PATH = "/images/";

private void writeImage(PhaseEvent event, String resourceName) {
URL url = getClass().getResource(IMAGE_PATH + resourceName);
URLConnection conn = null;
InputStream stream = null;
HttpServletResponse response =
(HttpServletResponse)event.getFacesContext().getExternalContext().getResponse();
try {
conn = url.openConnection();
conn.setUseCaches(false);
stream = conn.getInputStream();
ServletContext servletContext =
(ServletContext)FacesContext.getCurrentInstance().getExternalContext().getContext();

String mimeType = servletContext.getMimeType(resourceName);
response.setContentType(mimeType);
response.setStatus(200);

// Copy the contents of the file to the output stream
byte[] buf = new byte[1024];
int count = 0;
while ((count = stream.read(buf)) >= 0) {
response.getOutputStream().write(buf, 0, count);
}
response.getOutputStream().close();

} catch (Exception e) {
String message = null;
message = "Can't load image file:" + url.toExternalForm();
try {
response.sendError(HttpServletResponse.SC_BAD_REQUEST,
message);
} catch (IOException f) {
f.printStackTrace();
}
}
}
}
```

PhaseListeners need to be configured in order to be active. This configuration usually is done in the faces-config.xml of the application. Fortunately, we can also configure the PhaseListener in the faces-config.xml file that we create for the custom component. This faces-config.xml is part of the jar file in which the custom component is shipped and deployed. Its contents are merged with the application's own faces-config.xml. The registration of our

PhaseListener looks like this:

```
<faces-config version="1.2" xmlns="http://java.sun.com/xml/ns/javaee">
<lifecycle>
<phase-listener>nl.amis.jsf.ResourceServerPhaseListener</phase-listener>
</lifecycle>
<component>
   ...
```

Triggering events on the custom component

Time to take another big step. We will support clicking the image by the end user and turn that event into a reshuffle of the facets of the Shuffler component. In the next section we will not only act on that click ourselves, but also publish an event that others can listen to.

We will have to add a JavaScript event listener in the HTML rendered for the Shuffler. This client side code is triggered when the image is clicked. It will submit the form - after it has added an input element to the DOM and set a value on it. Note: this approach to have a custom component trigger an event that can be received by the server side renderer class has been described in Pro JSF and Ajax: Building Rich Internet Components - by John R. Fallows and Jonas Jacobi, the guys who first introduced me to JavaServer Faces.

The JavaScript for the Shuffler component looks like this:

```
/**
* The onclick handler for ShufflerRenderer.
*
* @param formClientId  the clientId of the enclosing UIForm component
* @param clientId      the clientId of the Shuffler component
*/
function _shuffle_click( formClientId, clientId)
{
var form = document.forms[formClientId];
var input = form[clientId];
if (!input) // if the input element does not already exist, create it and add it to the form
{
input = document.createElement("input");
input.type = 'hidden';
input.name = clientId;
form.appendChild(input);
}
input.value = 'clicked';
form.submit();
}
```

The JavaScript is not be directly included in the page - as it is part of the jar file in which the Shuffler component is shipped. We need a way to attach this JavaScript (it is in a file called shuffle.js) to the page from within the custom component, or in this case rather its Renderer class. We extend the ResourceServerPhaseListener to also handle JavaScript resources, just like it can handle images.

```
public class ResourceServerPhaseListener implements PhaseListener {
public static final String RENDER_SCRIPT_TAG = "/js/";
public static final String RENDER_IMAGE_TAG = "/images/";
public static final String SCRIPT_PATH = "/js/";
public static final String IMAGE_PATH = "/images/";

public PhaseId getPhaseId() {
return PhaseId.RESTORE_VIEW;
}

public void afterPhase(PhaseEvent event) {
// If this is restoreView phase
if (PhaseId.RESTORE_VIEW == event.getPhaseId()) {

// if the request is for a JavaScript library
if (-1 != event.getFacesContext().getViewRoot().getViewId().indexOf(RENDER_SCRIPT_TAG)) {
// extract the name of the script from the ViewId
String script =
event.getFacesContext().getViewRoot().getViewId().substring(event.getFacesContext()
.getViewRoot().getViewId().indexOf(RENDER_SCRIPT_TAG) +
RENDER_SCRIPT_TAG.length());
// render the script
writeScript(event, script);
event.getFacesContext().responseComplete();
}
...  image handling, same as before
}
}

public void beforePhase(PhaseEvent event) {
}

private void writeScript(PhaseEvent event, String resourceName) {
URL url = getClass().getResource(SCRIPT_PATH + resourceName);
URLConnection conn = null;
InputStream stream = null;
BufferedReader bufReader = null;
HttpServletResponse response =
(HttpServletResponse)event.getFacesContext().getExternalContext().getResponse();
OutputStreamWriter outWriter = null;
String curLine = null;

try {
outWriter =
new OutputStreamWriter(response.getOutputStream(), response.getCharacterEncoding());
conn = url.openConnection();
conn.setUseCaches(false);
stream = conn.getInputStream();
bufReader = new BufferedReader(new InputStreamReader(stream));
response.setContentType("text/javascript");
response.setStatus(200);
while (null != (curLine = bufReader.readLine())) {
outWriter.write(curLine + "\n");
}
outWriter.close();
} catch (Exception e) {
String message = null;
message = "Can't load script file:" + url.toExternalForm();

try {
response.sendError(HttpServletResponse.SC_BAD_REQUEST,
message);
} catch (IOException f) {
f.printStackTrace();
}
}
}

private void writeImage(PhaseEvent event, String resourceName) {
... same as before
}
}
```

The Renderer class is responsible for rendering the markup that will include the JavaScript resources to the page (the script element). We could have multiple occurrences of our custom component in a page. However, the JavaScript file shuffle.js should be loaded only once, to prevent excessive and completely pointless browser requests. In order to make that happen, the Renderer indicates to a method writeScriptResource that it has a JavaScript resource that should be included. This method verifies whether a script tag for downloading that same resource has already been added in the current request. If so, it will not add another script tag. If not [already included] then the tag is added with its src attribute referring to the proper PhaseListener controlled url:

```
protected void writeScriptResource(
FacesContext context,
String      resourcePath) throws IOException
{
Set scriptResources = _getScriptResourcesAlreadyWritten(context);

// Set.add() returns true only if item was added to the set
// and returns false if item was already present in the set
if (scriptResources.add(resourcePath))
{
ViewHandler handler = context.getApplication().getViewHandler();
String resourceURL = handler.getResourceURL(context, SCRIPT_PATH +resourcePath);
ResponseWriter out = context.getResponseWriter();
out.startElement("script", null);
out.writeAttribute("type", "text/javascript", null);
out.writeAttribute("src", resourceURL, null);
out.endElement("script");
}
}

private Set _getScriptResourcesAlreadyWritten(
FacesContext context)
{
ExternalContext external = context.getExternalContext();
Map requestScope = external.getRequestMap();
Set written = (Set)requestScope.get(_SCRIPT_RESOURCES_KEY);

if (written == null)
{
written = new HashSet();
requestScope.put(_SCRIPT_RESOURCES_KEY, written);
}

return written;
}

static private final String _SCRIPT_RESOURCES_KEY =
ShufflerRenderer.class.getName() + ".SCRIPTS_WRITTEN";
```

With these helper methods in place, the ShufflerRenderer can be extended to include the client side click handling code:

```
@Override
public void encodeBegin(final FacesContext facesContext,
final UIComponent component) throws IOException {
super.encodeBegin(facesContext, component);
final Map<String, Object> attributes = component.getAttributes();
final ResponseWriter writer = facesContext.getResponseWriter();
String formClientId = _findFormClientId(facesContext, component);
String shuffleClientId = component.getClientId(facesContext);

<b>writeScriptResource(context, "shuffle.js");
</b>writer.startElement("DIV", component);
String styleClass =
(String)attributes.get(Shuffler.STYLECLASS_ATTRIBUTE_KEY);
writer.writeAttribute("class", styleClass, null);
<b>writer.startElement("SPAN", component);
writer.writeAttribute("onClick",
"_shuffle_click('" + formClientId +
"','" + "'" + shuffleClientId + "')", null);</b>
writer.startElement("IMG", component);
writer.writeAttribute("src", imageUrl( facesContext,SHUFFLE_IMAGE), null);
writer.writeAttribute("alt", "Click to reshuffle", null);
writer.writeAttribute("width", "20px", null);
writer.endElement("IMG");
<b>        writer.endElement("SPAN");</b>

List<UIComponent> orderedFacets = ((Shuffler)component).getOrderedFacets(facesContext);
for (UIComponent facet:orderedFacets) {
facet.encodeAll(facesContext);
}
}

protected void encodeResources(FacesContext context,
UIComponent component) throws IOException {
writeScriptResource(context, "shuffle.js");
}

/**
* Finds the parent UIForm component client identifier.
*
* @param context    the Faces context
* @param component  the Faces component
*
* @return  the parent UIForm or RichForm (for usage in ADF) client identifier, if present, otherwise null
*/
private String _findFormClientId(FacesContext context,
UIComponent component) {
if (component==null) {
return null;
}
if (component instanceof UIForm || component.getClass().getName().endsWith("RichForm")) {
return component.getClientId(context);
}
else {
return _findFormClientId(context, component.getParent());
}

}
```

The image is wrapped in a SPAN and the onclick event handler is defined on that SPAN element (this allows us to later on add more clickable stuff to the SPAN). When the image is clicked, the _shuffle_click function is invoked - that was loaded from shuffle.js. The element is added to the form and the form is submitted.

The HTML rendered from this renderer now looks like this:

```
<script src="/CustomJSFConsumer-ViewController-context-root/faces/js/shuffle.js"type="text/javascript">
<div class="h1">
<span onclick="_shuffle_click('f1','s1')">
<img width="20" alt="Click to reshuffle" src="/CustomJSFConsumer-ViewController-context-root/faces/images/shuffleIcon.png"/>
</span>
```

```
... content from the facets
</div>
```

When the user clicks on the image, the following request is sent to the server:

javax.faces.ViewState	**!-osretn5tu**
org.apache.myfaces.trinidad.faces.FORM	f1
s1	clicked

This request is processed through the JSF lifecycle. The ShufflerRenderer should implement the decode() method to determine whether the s1 element is in the request, with the value clicked. If it finds that this is the case, it has established that the user has indeed clicked on the shuffle image, and appropriate action should take place. Here is the code for that decode() implementation:

```
@Override
public void decode(FacesContext facesContext, UIComponent component) {
ExternalContext external = facesContext.getExternalContext();
Map requestParams = external.getRequestParameterMap();

String clientId = component.getClientId(facesContext);
String clicked = (String)requestParams.get(clientId);

if (clicked != null && clicked.length() > 0 &&
"clicked".equalsIgnoreCase(clicked)) {
((Shuffler)component).notifyOfShuffleEvent(facesContext);
} //if
} //decode
```

This code calls upon the Shuffler component to handle the event. The notifyOfShuffleEvent() method in the Shuffler component is implemented (for now at least) in simple way:

```
void notifyOfShuffleEvent(FacesContext facesContext) {
String facetOrder =
(String)getAttributes().get(Shuffler.FACETORDER_ATTRIBUTE_KEY);
facetOrder = "reverse".equalsIgnoreCase(facetOrder)?"normal":
("normal".equalsIgnoreCase(facetOrder)?"reverse":facetOrder);
getAttributes().put(Shuffler.FACETORDER_ATTRIBUTE_KEY, facetOrder);
}
```

Later on we will have this method instantiate and broadcast a FacesEvent and invoke a directly registered listener.

When the page is rerendered, the facetOrder of the component has been swapped from normal to reverse or vice versa or it has stayed at random. In all cases, rerendering the page will have the effect of reshuffling the contents of the Shuffler component.

Publishing a JSF event and supporting event listeners

When the shuffle (click) event has occurred and has been established by the ShufflerRenderer that in turn has notified the Shuffler component that takes appropriate action, it is very well possible that other interested parties would like to know about the event as well. A first step in the direction of publishing an event to interested parties is the introduction of a single attribute shuffleListener, a methodExpression that can be used to configure a listener method (very much like the valueChangeListener and actionListener attributes on input components and action components respectively. This attribute can be used as follows:

```
<shf:shuffler styleClass="h1" facetOrder="random" id="s1"
shuffleListener="#{bean.shuffleEventHandler}">   ...
```

where the method shuffleEventHandler is implemented as follows:

```
public void shuffleEventHandler(Shuffler source, String comment) {
System.out.println("Shuffle Event received - with message "+comment );
}
```

which is extremely unexciting of course. However, this method can be extended to create a List in some creative way based on the list of facets that is passed in.

In order to add this methodExpression type of attribute, we have to go through the same steps we saw before:

- adding an attribute entry in the TLD
- adding support for processing the attribute in the Tag Handler (a setter and a line of code in setProperties())
- adding the attribute in saveState() and restoreState() in the Component class

TLD entry:

```
...
<attribute>
<name>shuffleListener</name>
<required>false</required>
<deferred-method>
<method-signature>void listener(nl.amis.jsf.shuffler.Shuffler ,
java.lang.String)</method-signature>
</deferred-method>
</attribute>
</tag>
```

Tag Handler:

```
private MethodExpression shuffleListener;
protected void setProperties(UIComponent component) {
super.setProperties(component);
processProperty(component, styleClass, Shuffler.STYLECLASS_ATTRIBUTE_KEY);
processProperty(component, facetOrder, Shuffler.FACETORDER_ATTRIBUTE_KEY);
((Shuffler)component).setShuffleListener( shuffleListener);    }

public void release() {
super.release();
styleClass= null;
facetOrder= null;
shuffleListener = null;    }
```

```
public void setShuffleProcessor(MethodExpression shuffleProcessor) {
this.shuffleProcessor = shuffleProcessor;    }
```

saveState() and restoreState() in Shuffler:

```
@Override
public Object saveState(FacesContext facesContext) {
Object values[] = new Object[4];
values[0] = super.saveState(facesContext);
values[1] = this.getAttributes().get(STYLECLASS_ATTRIBUTE_KEY);
values[2] = this.getAttributes().get(FACETORDER_ATTRIBUTE_KEY);
values[3] = this.shuffleListener;        return values;
}

@Override
public void restoreState(FacesContext facesContext, Object state) {
Object values[] = (Object[])state;
super.restoreState(facesContext, values[0]);
this.getAttributes().put(STYLECLASS_ATTRIBUTE_KEY, values[1]);
this.getAttributes().put(FACETORDER_ATTRIBUTE_KEY, values[2]);
this.setShuffleListener((MethodExpression)values[3]);}
```

Then we have to add the code in the Shuffler Component class that actually calls the method.

```
void notifyOfShuffleEvent(FacesContext facesContext) {
String facetOrder =
(String)getAttributes().get(Shuffler.FACETORDER_ATTRIBUTE_KEY);
facetOrder = "reverse".equalsIgnoreCase(facetOrder)?"normal":
("normal".equalsIgnoreCase(facetOrder)?"reverse":facetOrder);
getAttributes().put(Shuffler.FACETORDER_ATTRIBUTE_KEY, facetOrder);
if (getShuffleListener()!=null) {
try {
Object[] args = { this, "ShuffleEvent" };
getShuffleListener().invoke(facesContext.getELContext(), args);
}
catch (Exception e){}
}
}
```

This is one way to go about events and listeners. However, it is not the best way. For starters, we can only register a single listener in this fashion. We also have the event ‘published’ very early in the JSF lifecycle - during Apply Request Values (from the decode method). JavaServer Faces has a built in mechanism for dealing with events. We can leverage this standard facility in the following way:

- implement the ShuffleEvent class that extends from the FacesEvent interface
- create the ShuffleEventListener interface that extends from FacesListener
- define the shuffleListener tag entry in the tld
- implement the ShuffleEventListenerTag class to process shuffleEventListener tags in the jsp file
- add the addShuffleEventListener() method in the Shuffler component class that can be used from the ShuffleEventListenerTag to register a ShuffleEventListener on the component
- broadcast the ShuffleEvent through the JSF eventing infrastructure to all registered listeners
- With the new shuffleEventListeners, we can add as many interested parties to Shuffler as we care to:

```
<shf:shuffler styleClass="h1" facetOrder="random" id="s1"shuffleProcessor="#{shuffleBean.specialShuffle}"
shuffleListener="#{bean.shuffleEventHandler}">
<shf:shuffleListener type="view.ShuffleListener"/>
<shf:shuffleListener type="view.SomeOtherShuffleListener"/>
<shf:shuffleListener type="view.AndYetAnotherOneShuffleListener"/>
<f:facet name="1">
... content
</f:facet>
<f:facet name="2">
... content
</f:facet>
<f:facet name="3">
... content
</f:facet>
</shf:shuffler>
```

The class view.ShuffleListener implements the ShuffleEventListener interface that is defined along with the JSF component. This implementation can be as simple as:

```
package view;

import nl.amis.jsf.shuffler.ShuffleEvent;
import nl.amis.jsf.shuffler.ShuffleEventListener;

public class ShuffleListener implements ShuffleEventListener {
public void processEvent(ShuffleEvent shuffleEvent) {
System.out.println("The listener reports a shuffle event!");
}
}
```

when the user clicks on the shuffle image and the event is published, an instance of ShuffleListener is created and its processEvent() method is invoked by the JSF event broadcast system. In this simple case, the listener will do nothing but write a message to the system output.

The implementation according to the list of steps discussed before:

Implement the ShuffleEvent class that extends from the FacesEvent interface

```
package nl.amis.jsf.shuffler;

import javax.faces.component.UIComponent;
import javax.faces.event.FacesEvent;
import javax.faces.event.FacesListener;
import javax.faces.event.PhaseId;

public class ShuffleEvent extends FacesEvent {
public ShuffleEvent(UIComponent source) {
super(source);
setPhaseId(PhaseId.INVOKE_APPLICATION);
}

public boolean isAppropriateListener(FacesListener facesListener) {
return (facesListener instanceof ShuffleEventListener);
}

public void processListener(FacesListener facesListener) {
((ShuffleEventListener)facesListener).processEvent(this);
}
}
```

Create the ShuffleEventListener interface that extends from FacesListener

```
package nl.amis.jsf.shuffler;

import javax.faces.event.FacesListener;

public interface ShuffleEventListener extends FacesListener{

public void processEvent( ShuffleEvent event) ;
}
```

TLD entry:

```
....
<tag>
<name>shuffleListener</name>
<tag-class>nl.amis.jsf.shuffler.ShuffleEventListenerTag</tag-class>
<body-content>empty</body-content>
<attribute>
<description>
The fully qualified class name for the shuffle event listener.
</description>
<name>type</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
</tag>
```

The Tag Handler class ShuffleEventListenerTag- note that it extends TagSupport rather than UIComponentELTag. That is because in this case we do not want a proper JSF component to be added to the View tree based on the shuffleListener tags. For each tag, we will register a listener on the parent Shuffler component - that is our integration point with JSF in this case

```
package nl.amis.jsf.shuffler;

import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentClassicTagBase;
import javax.faces.webapp.UIComponentELTag;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;

public class ShuffleEventListenerTag extends TagSupport {

String shuffleEventListenerType;

/**
* @return SKIP_BODY, always
* @throws JspException  if an error condition occurs
*/
public int doStartTag() throws JspException {
UIComponentClassicTagBase tag =
UIComponentELTag.getParentUIComponentClassicTagBase(pageContext);
if (tag == null)
throw new JspException("Not inside UIComponentTag");

if (tag.getCreated()) {
UIComponent component = tag.getComponentInstance();
if (component == null)
throw new JspException("Component instance is null");
if (!(component instanceof Shuffler))
throw new JspException("Component is not a Shuffler");

Shuffler shuffler = (Shuffler)component;
String className = shuffleEventListenerType;
ShuffleEventListener listener =
createShuffleEventListener(className);
shuffler.addShuffleEventListener(listener);
}
return (SKIP_BODY);
}

/**
* Sets the fully qualified class name of the
* shuffleEventListenerType instance to be created.
*
* @param type  the class name
*/
public void setType(String type) {
shuffleEventListenerType = type;
}

protected ShuffleEventListener createShuffleEventListener(String className) throwsJspException {
try {
ClassLoader loader =
Thread.currentThread().getContextClassLoader();
Class clazz = loader.loadClass(className);
return ((ShuffleEventListener)clazz.newInstance());
} catch (Exception e) {
throw new JspException(e);
}
}
}
```

Add the addShuffleEventListener() method in the Shuffler component class that can be used from the ShuffleEventListenerTag to register a ShuffleEventListener on the component

```
public void addShuffleEventListener(ShuffleEventListener listener) {
addFacesListener(listener);
}
```

Broadcast the ShuffleEvent through the JSF eventing infrastructure to all registered listeners

```
void notifyOfShuffleEvent(FacesContext facesContext) {
String facetOrder =
(String)getAttributes().get(Shuffler.FACETORDER_ATTRIBUTE_KEY);
facetOrder = "reverse".equalsIgnoreCase(facetOrder)?"normal"
:("normal".equalsIgnoreCase(facetOrder)?"reverse":facetOrder);
getAttributes().put(Shuffler.FACETORDER_ATTRIBUTE_KEY, facetOrder);
if (getShuffleListener()!=null) {
try {
Object[] args = { this, "ShuffleEvent" };
getShuffleListener().invoke(facesContext.getELContext(), args);
}
catch (Exception e){}
}
// queue the event to be broadcast at the indicated phase
// (inside the ShuffleEvent, invoke application) to the registered listeners
ShuffleEvent shuffleEvent = new ShuffleEvent(this);
shuffleEvent.queue();
}
```

Registration of a custom shuffle-processor MethodExpression

Sometimes we may want to pass a MethodExpression in one of the attributes on a custom JSF component. For example to provide the component with a method it can call in order to pre or post process some values. In our Shuffler example, we will add a shuffleProcessor attribute. This attribute can be set to a methodExpression that refers to a method with a specific signature: java.util.List processor(java.util.List). That means: a method that a List as an input parameter and that returns a List as result. This method will be called by the Shuffler to have the list of facets re-ordered by the externally supplied method that may implement other shuffle patterns besides normal, reverse and random. This attribute can be used as follows:

```
<shf:shuffler styleClass="h1" facetOrder="random" id="s1"
shuffleProcessor="#{shuffleBean.specialShuffle}"
shuffleListener="#{bean.shuffleEventHandler}">
...
```

where the method specialShuffle is implemented as follows:

```
public List<UIComponent> specialShuffle( List<UIComponent>  facetList){
return facetList;
}
```

which is extremely unexciting of course. However, this method can be extended to create a List in some creative way based on the list of facets that is passed in.

In order to add this methodExpression type of attribute, we have to go through the same steps we saw before:

- adding an attribute entry in the TLD
- adding support for processing the attribute in the Tag Handler (a setter and a line of code in setProperties())
- adding the attribute in saveState() and restoreState() in the Component class

TLD entry:

```
...
<attribute>
<name>shuffleProcessor</name>
<required>false</required>
<deferred-method>
<method-signature>java.util.List processor(java.util.List)</method-signature>
</deferred-method>
</attribute>
</tag>
```

Tag Handler:

```
private MethodExpression shuffleProcessor;
protected void setProperties(UIComponent component) {
super.setProperties(component);
processProperty(component, styleClass, Shuffler.STYLECLASS_ATTRIBUTE_KEY);
processProperty(component, facetOrder, Shuffler.FACETORDER_ATTRIBUTE_KEY);
((Shuffler)component).setShuffleListener( shuffleListener);
((Shuffler)component).setShuffleProcessor( shuffleProcessor);    }

public void release() {
super.release();
styleClass= null;
facetOrder= null;
shuffleListener = null;
shuffleProcessor = null;    }

public void setShuffleProcessor(MethodExpression shuffleProcessor) {
this.shuffleProcessor = shuffleProcessor;
}
```

saveState() and restoreState() in Shuffler:

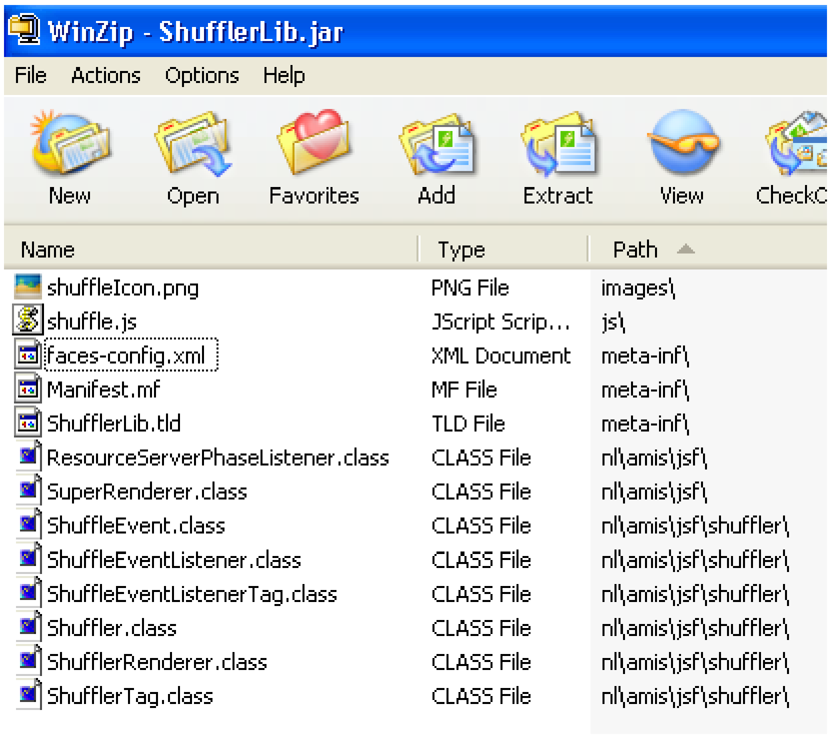
```
@Override
public Object saveState(FacesContext facesContext) {
Object values[] = new Object[5];
values[0] = super.saveState(facesContext);
values[1] = this.getAttributes().get(STYLECLASS_ATTRIBUTE_KEY);
values[2] = this.getAttributes().get(FACETORDER_ATTRIBUTE_KEY);
values[3] = this.shuffleListener;
values[4] = this.shuffleProcessor;        return values;
}

@Override
public void restoreState(FacesContext facesContext, Object state) {
Object values[] = (Object[])state;
super.restoreState(facesContext, values[0]);
this.getAttributes().put(STYLECLASS_ATTRIBUTE_KEY, values[1]);
this.getAttributes().put(FACETORDER_ATTRIBUTE_KEY, values[2]);
this.setShuffleListener((MethodExpression)values[3]);
this.setShuffleProcessor((MethodExpression)values[4]);    }
```

Then we have to add the code in the Shuffler Component class that actually calls the method.

```
public List<UIComponent> getOrderedFacets(FacesContext facesContext) {
... same as before
// if a shuffleProcessor is configured, we need to invoke it to provide us with the list of facets in the order in which to render them
// not that strictly speaking the shuffleProcessor can decide to leave out certain facets for whatever reason
if (shuffleProcessor != null) {
try {
Object[] args = { orderedFacets };
orderedFacets =
(List<UIComponent>)shuffleProcessor.invoke(facesContext.getELContext(),
args);
} catch (Exception e) {
}
}
return orderedFacets;
}
```

Deploying the Custom Component

The custom component is deployed in a JAR file - just like for example any other bunch of custom JSP tags.Use your favorite build tool for constructing the JAR file. It should look something like this:



Name	Type	Path
shuffleIcon.png	PNG File	images\
shuffle.js	JScript Scrip...	js\
faces-config.xml	XML Document	meta-inf\
Manifest.mf	MF File	meta-inf\
ShufflerLib.tld	TLD File	meta-inf\
ResourceServerPhaseListener.class	CLASS File	nl\amis\jsf\
SuperRenderer.class	CLASS File	nl\amis\jsf\
ShuffleEvent.class	CLASS File	nl\amis\jsf\shuffler\
ShuffleEventListener.class	CLASS File	nl\amis\jsf\shuffler\
ShuffleEventListenerTag.class	CLASS File	nl\amis\jsf\shuffler\
Shuffler.class	CLASS File	nl\amis\jsf\shuffler\
ShufflerRenderer.class	CLASS File	nl\amis\jsf\shuffler\
ShufflerTag.class	CLASS File	nl\amis\jsf\shuffler\

Consume the Custom JSF Component:
In your JSF 1.2 web application, add the JAR file created during deployment as a tag library. To include the Shuffler in a page, add components from the tag-library to a JSF page (the taglib';s namespace should be registered in the header of the JSP(X) file) and configure its attributes, facets and listeners
A snippet from a page that is using the Shuffler:

```
<?xml version='1.0' encoding='windows-1252'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:af="http://xmlns.oracle.com/adf/faces/rich"
xmlns:shf="/nl.amis.jsf/ShufflerLib">
<jsp:directive.page contentType="text/html;charset=windows-1252"/>
<f:view>
<af:document id="d1">
<af:form id="f1">
<af:panelBox text="PanelBox1" id="pb1">
<shf:shuffler styleClass="h1" facetOrder="random" id="s1"shuffleProcessor="#{shuffleBean.specialShuffle}"
shuffleListener="#{bean.shuffleEventHandler}">
<f:facet name="1">
<af:commandButton text="commandButton 1" id="cb1"/>
</f:facet>
<f:facet name="3">
<af:commandButton text="commandButton 2" id="cb2"/>
</f:facet>
...
```

```
<shf:shuffleListener type="view.ShuffleListener"/>
</shf:shuffler>
</af:panelBox>
</af:form>
</af:document>
</f:view>
</jsp:root>
```

In this example, the application has registered a bean.shuffleEventHandler method expression (on the attribute shuffleListener), a shuffleBean.specialShuffle method expression (on the attribute shuffleProcessor) and a shuffleListener event listener child component. These methods and the view.ShuffleListener have to be implemented - as was already discussed earlier in this article.