Custom Components

Agenda

- What are Custom Components
- Why Custom Components?
- Important JSF API Classes
- Custom Component Building Blocks
- Writing the Custom Component Class
- Encoding
  - Using the ResponseWriter
- Decoding
  - Writing the Tag Class
  - Building the TLD
- Configuring the Custom Componen

Introduction

- One of the key strengths of JavaServer Faces (JSF) is that not only does it provide substantial technology for easy, out of the box component based J2EE Web applications assembly, but it also is a very flexible API which allows for a wide breadth of customizations in numerous and innovative ways

- Reference implementation of JSF from Sun Microsystems provide only a hand full of JSF Components

- The commercial implementation of GUI Tools for Java Server Faces such as WebSphere Application Developer and Java Studio Creator do come with significant additions to the set of components that allows creating interfaces similar to that of Swing applications
- We can also create UI components like date Picker, calendar, Editable tables and many more using JSF API.

What are custom components?

- A JSF UI component which is not provided by sun JSF specification but can be created using JSF API, which provides customized behavior and rendering as required by the user
  - E.g Tomahawk library from apache myfaces implementation provides a number of custom components.
  - Oracle ADF faces has lot more custom components.
  - ICE faces is another implementation which comes with a number of AJAX Enabled JSF custom components.
  - There are many others.

When to Use Custom Component

- Need to add a new behavior to a standard component
- Need to aggregate components
- Need a complex HTML component that is not supported by JSF implementation
- Need to render to a non-HTML client

Decoding & Encoding by a Component

- If you are creating a custom component, that your component class performs these two operations:
  - Decoding: Converting the incoming request parameters to the local value of the component
  - Encoding: Converting the current local value of the component into the corresponding markup that represents it in the response

Two Ways of Decoding/Encoding

- JSF specification supports two programming models for handling encoding and decoding:
  - Direct implementation: The component class itself implements the decoding and encoding
  - Delegated implementation: The component class delegates the implementation of encoding and decoding to a separate renderer

- You can use both during runtime
  - Your component class can include some default rendering code, but it can delegate rendering to a renderer if there is one.

When to Use Custom Renderer?

- To associate your component with different renderers so that you can represent the component in different ways on the page
  - If you don't plan to render a particular component in different ways, it's simpler to let the component class handle the rendering.

Creating a Custom component

Parts of a JSF Custom Component

- **UIComponent Class**
  - A Java class derived from either the UIComponentBase or extended from an existing JSF UIComponent such as outputText.
  - This is the actual Java class representing the core logic of the component. It can optionally contain the logic to "render" itself to a client, or rendering logic can be separated into a separate "renderer" class.
- **Renderer Class**
  - This is a class that only contains code to render a UIComponent.
  - Rendering classes can provide different renderings for a UI Component.
  - For such as either a button or hyperlink for the UICommand component.
  - Renderers can also provide different rendering options for the same UI Component on **different** client types such as browsers, PDAs etc.
  - This allows JSF the ability to run on any client device providing a corresponding set of renderer classes are built for the specific client type.

Parts of a JSF Custom Component

- **UI Component Tag Class**
  - This is a JSP tag handler class that allows the UI Component to be used in a JSP. It can also associate a separate renderer class with a UIComponent class. (Note: The UI Component Tag handler class is only required for usage in a JSP deployment environment. Keep in mind that UI Components can also be used in non-JSP environments.)
- **Tag Library Descriptor File**
  - This is a standard J2EE JSP tag library descriptor (tld) file which associates the tag handler class with a usable tag in a JSP page. (Required only For JSP usage only.)
- **Associated helper classes**
  - These include a collection of standard (included in JSF RI) or custom helper classes such as Converters, Validators, ActionListeners etc.

Steps to create a custom component

- Write the UI Component class
  - The component can render itself also
- Write the Renderer class (Optional)
  - if rendering is delegated to a Renderer
- Write the Tag class
  - This class is also called a tag handler
- Write the tag library handler (.TLD)
- Configure the custom component in faces-config.xml
- Use the tag in a JSP

Write the UI Component class

- A component class defines state and behavior of a UI component
- State
  - Type, Identifier, Local value
- Behavior to support
  - Decoding and encoding
  - Saving state of component
  - Updating bean value with local value
  - Processing validation on the local value
  - Queuing events

Which Base Class to Extend?

- Two choices
  - Extend UIComponentBase directly
  - Extend a existing standard component

- Whenever possible, extend a existing standard component
  - For example, editable menu component can be extended from UISelectOne

Behavioral Interfaces (page 1)

- *ActionSource*
  - component can fire ActionEvent
- *ValueHolder*
  - component maintains a local value as well as the option of accessing data in model tier
- *EditableValueHolder*
  - extends ValueHolder
  - additional features for editable components such as validation and emitting value-change events

Behavioral Interfaces (page 2)

- *NamingContainer*
  - component has a unique ID
- *StateHolder*
  - component has state that must be saved between requests

Behavioral Interfaces (page 3)

- *NamingContainer*
  - component has a unique ID
- *StateHolder*
  - component has state that must be saved between requests

The UIComponent Class

```
public class MyComp extends UIOutput {

public void encodeBegin(FacesContext context) throws IOException {
        super.encodeBegin(context);
        String style="background-color:blue;width:100px;height:200px";
        ResponseWriter writer = context.getResponseWriter();
        writer.startElement("div",this);
        writer.writeAttribute("id", getClientId(context), null);
        writer.writeAttribute("style", style, null);

    }

public void encodeEnd(FacesContext context) throws IOException {
        super.encodeEnd(context);
        ResponseWriter writer= context.getResponseWriter();
        writer.endElement("div");
    }
    }
```

Write the Renderer class

- We do not a Renderer as the rendering code is written inside the component class
- The component class can render tself

Write the Tag class

- Tag handler drives render response phase
- Handles
  - retrieving component type
  - returning renderer type (if a separate render is used for rendering)
- returns null if the component handles the rendering
- setting component attributes to the values given in the page

The Tag class (tag handler)

```
public class MyCompTag extends UIComponentTag{

protected void setProperties(UIComponent arg0) {
        super.setProperties(arg0);
    }

public String getComponentType() {
        return "mycomp";
    }

public String getRendererType() {
        return null;
    }

  }
```

Write the tag library handler (.TLD)

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-
    jsptaglibrary_2_0.xsd">
  <tlib-version>1.0</tlib-version>
  <short-name>mycomp.tld</short-name>
  <uri>http://sdgcorp.com/</uri>
  <tag>
      <name>test</name>
      <tag-class>com.jp.jsf.MyCompTag</tag-class>
      <body-content>JSP</body-content>
  </tag>
</taglib>
```

Configure the custom component in faces-config.xml (page1)

```
<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">

<component>
    <component-type>mycomp</component-type>
    <component-class>com.jp.jsf.MyComp</component-class>
</component>
</faces-config>
```

Configure the custom component in faces-config.xml (page1)

```
<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">

<component>
    <component-type>mycomp</component-type>
    <component-class>com.jp.jsf.MyComp</component-class>
</component>
</faces-config>
```

Configure the custom component in faces-config.xml (page2)

- This renderer is required when you write your own renderer class

```
<faces-config>
<component>
  <component-type>mydiv</component-type>
  <component-class>javax.faces.component.UIOutput</component-class>
 </component>

 <render-kit>
<renderer>
<component-family>javax.faces.Output</component-family>
<renderer-type>renderer</renderer-type>
<renderer-class>com.jp.jsf.MyRenderer</renderer-class>

</renderer>

</render-kit>
</faces-config>
```

Use the tag in a JSP

```
<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@ taglib  uri="http://sdgcorp.com/"  prefix="m"%>
<html>
    <head>
        <title>JSP Page</title>
    </head>
    <body>
        <f:view>
            <h1><h:outputText value="JavaServer Faces" /></h1>
            <m:test></m:test>
        </f:view>
    </body>
</html>
```

Thank You