CreatingClayComponents

Creating Clay components

There are numerous ready-made components within the Java{{`Server Faces world. These are available in Clay too (There is a tool that maps the tag library description;tld into its Clay component counterpart call Tld2}}`Clay`Cfg), so there is normally not a big need for creating own components (from scratch). But in many cases you may not be satisfied with what the standard components have to offer or you simply want to adapt them to your own special needs or not to forget add basic components together to form a new reusable component.

In this tutorial we look at some components that we want to create for our site (we will be making some sub-components too) , amongst others: a person registration form, a news component and a table/list.

If don't want to tag along and enter code as we go, you can download the completed stuff shaleclay2.zip

Start by opening up the Shale/Clay project that we created using the Maven2 archetype in the this tutorial

A person registration form

This component will be used to enter information about a person. The first thing we do is to define a bean that will be responsible for holding the entered information. The simplest way to do this is to augment the Person bean from the archetype.

Add the following attributes:

- lastname - String
- firstname - String
- streetadress - String
- post - Post = new Post()

Create getters/setters for all.

Then create the Post bean with following attributes:

- zipcode - Short
- city - String
- zipcodes - Integer[] = new Integer[] {51000, 51001, 51002, 51020, 51030, 51050, 51100, 51200}

Create getters/setter for except for the zipcodes which only require a getter (read-only).

If you started with the Maven2 archetype, the person bean will already have been defined as a managed bean in the faces-config.xml file, if not enter this into it:

```
<managed-bean id="person">
    <managed-bean-name>person</managed-bean-name>
    <managed-bean-class>
        com.acme.test.Person
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Now we almost have the receiver for the data in place. What we lack is a viewcontroller (controller in MVC) for the person registration form. Create a bean PersonVC that extends Abstract{{`View}}`Controller and has one attribute of type Person with respective getter/setter.

Define the following in faces-config.xml file

```
<managed-bean id="personreg">
    <managed-bean-name>personreg</managed-bean-name>
    <managed-bean-class>
        com.acme.test.PersonVC
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
        <property-name>person</property-name>
        <property-class>
            com.acme.test.Person
        </property-class>
        <value>#{person}</value>
    </managed-property>
</managed-bean>
```

The next we do is to add a method that "saves" the person. Do this by adding the following method:

```
public String save()
{
    return null;
}
```

For now this method does nothing. Also note that it returns null. This means that no navigation rule will be executed and we remain at the same form.

Now the time has come to design the form it self, the way we want it. By that we mean how it will look when we have completed our person registration component. We do this so that we have something to compare our component against. This also simulates the situation where we have been handed a html mockup from our designers, and we are now to create a dynamic implementation of it. In our case we want it to look like this:

ShaleClay9.png!

As we can see from the design, there are two distinct blocks of information here, and would be natural to break this down into two components first. By so doing we can also reuse these by them self in another case. When we take a second look at it we see that there is a repetition of equal components; 2 labels and 2 input fields. We could of course created a component that was a combination of firstname label and firstname input field, but it would be cumbersome over time to have to create new components for different attributes (lastname, streetadress etc.). In some cases that is desirable though to ensure consistency everywhere (We will be creating such a component too). Instead we create a generic label/input combination. This we can then reuse for any attribute. We have two alternatives for how we create it, as an XML definition in the Clay configuration file, or as a HTML mockup with Clay annotations. To demonstrate both we will use the XML variant for the name panel and HTML for the address panel. We will start with the name panel, so let try to define it.

We start by creating the generic labell/input filed component and we will call it "gltic". This is short for Generic Label Text Input Component.

```
<component jsfid="gltic" extends="clay" id="gltic">
    <element jsfid="outputLabel" renderId="1">
        <attributes>
            <set name="value" value="@ltName"></set>
        </attributes>
    </element>
    <element jsfid="inputText" renderId="2">
        <attributes>
            <set name="value" value="@ltValue"></set>
        </attributes>
    </element>
</component>
```

There are a couple of things to note here. Firstsly, we extend from "clay". All components must either extend "clay" or another existing component. Next thing to note is that within the component define elements that are part of the component. Be careful with the rendered attribute. This attribute governs the rendering order of the elements to the device. So be careful to order them in the order that you want them displayed. Also take care that the renderIDs are unique at the same element level (You can have nested elements).

As you can see we have tied the values of the attributes to symbols (identified by @). This means that we do not have to worry about the actual values of these now, but handle that when we use the component.

Lets now try to define the first panel, which is the name panel. As we said earlier there are a lot of ready-made component libraries for us to choose from. We are now going to use one of these. It is know as Tomahawk and comes from the MyFaces project. The component that we are going to use is the htmlTag. This is a component that will render any HTML tag that you specify as a value. To be able to use this, you need to download it and install it into you project. How you do this depends on whether you are using the Maven2 plugin, Maven2 by it self or not at all. The important thing is that you get the tomahawk jar file in the WEB-INF/LIB when you deploy your application(if it's a standalone WAR). We base our application on the latest version which is currently 1.5-SNAPSHOT. Next is to make Clay aware of the components. You do that by adding the tomahawk-1.1.5-SNAPSHOT-config.xml Clay configuration file to your WEB-INF folder. Then change the following section in the web.xml file:

```
    <!-- Clay Common Configuration Resources -->
    <context-param>
        <param-name>
            org.apache.shale.clay.COMMON_CONFIG_FILES
        </param-name>
        <param-value>
            /WEB-INF/clay-config.xml,
            /WEB-INF/tomahawk-1.1.5-SNAPSHOT-config.xml
        </param-value>
    </context-param>
```

We are now ready to use all the components that Tomahawk has to offer.

Back to our name panel, there is one thing that we need to decide. What to use as the outermost container? As you may know JSF builds a component tree and uses POST to send data from the client to the server and that requires a form. Since we are going to add more components to our page that carry data, we choose to use a tag that is frequently (container) used within a form: Fieldset. So this will be our container. Now our componet looks likes this:

```
        <component jsfid="namepanel" extends="clay" id="namepanel">
                <element jsfid="t:htmlTag" renderId="1" id="namefieldset">
                        <attributes>
                                <set name="value" value="fieldset" />
                        </attributes>
                        <element jsfid="t:htmlTag" renderId="1" id="namelegend">
                                <attributes>
                                        <set name="value" value="legend" />
                                </attributes>
                                <element jsfid="outputText" renderId="1">
                                        <attributes>
                                                <set name="value"
                                                        value="#{messages['namepanel.text']}">
                                                </set>
                                        </attributes>
                                </element>
                        </element>
                        <element jsfid="gltic" renderId="2" id="firstname">
                                <symbols>
                                        <set name="ltName"
                                                value="#{messages['firstname.label']}">
                                        </set>
                                        <set name="ltValue"
                                                value="#{@managed-bean-name.person['firstname']}">
                                        </set>
                                </symbols>
                        </element>
                        <element jsfid="t:htmlTag" renderId="3" id="namebr">
                                <attributes>
                                        <set name="value" value="br" />
                                </attributes>
                        </element>
                        <element jsfid="gltic" renderId="4" id="lastname">
                                <symbols>
                                        <set name="ltName"
                                                value="#{messages['lastname.label']}">
                                        </set>
                                        <set name="ltValue"
                                                value="#{@managed-bean-name.person['lastname']}">
                                        </set>
                                </symbols>
                        </element>
                </element>
        </component>
```

Looking closer at this, we see that the first element we define is the Tomahwk component htmlTag, identified by "t:htmlTag". We give it a value of "fieldset" through the attributes section. Then we define a nested element where we use "htmlTag" again, only this time we give it a value of "legend". However "legend" requires a value, so we create a nested element where we the standard Clay "outputText" component. The value we get from the resourcebundle (messages) – Remember to add that value now: personpanel.text=Person data.

Then we define the next nested element within the namepanel, remembering to increment the renderId. This element will be our reusable label/input (gltic) component. This time we use it to show a label and input for firstname. Remember to add firstname.label=Firstname to the resourcebundle. We wire the input field to the attribute firstname in the person bean of the viewcontroller that controls this view (@managed-bean-name).

Then we repeat the same work, only this time with lastname (remember the rendered and the resourcebundle)

We now have a panel where we can enter firstname and lastname. But we also need a panel to enter the address information. As we mentioned earlier this time we define the component in HTML. The principals are the same, only now we are using HTML and are able to do preview on what we do. The panel then becomes (stripped):

```
            <fieldset>
                <legend>
                    <span jsfid="outputText" value="@adresslegend" allowbody="false">Adress</span>
                </legend>
                <label jsfid="outputLabel" value="@streetadressLabel" allowbody="false">
                    Streetadress
                </label>
                <input jsfid="inputText" value="@streetadressField" type="text" allowbody="false">
                <br>
                <label jsfid="outputLabel" value="@zipcodeLabel" allowbody="false">
                    Zipcode
                </label>
                <input jsfid="inputText" value="@zipcodeField" type="text" allowbody="false">
                <br>
                <label jsfid="outputLabel" value="@cityLabel" allowbody="false">
                    City
                </label>
                <input jsfid="inputText" value="@cityField" type="text" allowbody="false">
            </fieldset>
```

As you can see this is pure HTML with some special attributes on some of the HTML tags. These attributes are Clay directives. The attribute "jsfid" points at a Clay component defined in a Clay configuration file. Next is "value" where we which value the tag should have. Since this is to be a reusable component, we use the symbol instead of hardcoded values. The "SPAN" tag is used as a replacement for the cases where we want to simply output some text that is not related to any tag. We save this in the pages folder and give it a name of adresspanel.html.

Now that we have defined our HTML based Clay component, we need to tell Clay of its existence. We do that by adding the following in our clay-config.xml file;

```
    <component jsfid="adresspanel" extends="clay" id="adresspanel">
        <attributes>
            <set name="clayJsfid" value="/pages/adresspanel.html" />
        </attributes>
    </component>
```

Note that we here use a special attribute "clayJsfid". This is Clay's variant of including something (ala jsp:include). So here we import our HTML component and assign it a Clay id (jsfid).

So now that we have defined our two components that we are going to use, what remains is to stitch the whole thing together. Again we can either define them in HTML or XML. Here we will use the latter. There is one thing that we have not looked at yet, the button that needs to be pushed in order to submit the data to the server. If you want a common "look&feel" for this type of component the it makes sense to define them, such that you later on can use them "as is". An example of this would be:

```
    <component jsfid="saveButton" extends="commandButton">
        <attributes>
            <set name="value" value="#{messages['savebutton.label']}"></set>
            <set name="action" value="#{@managed-bean-name.@method}" ></set>
        </attributes>
    </component>
```

Let's assume that you always wanted this to have a font-size of 10px and a green backgroud and white bold font. The only thing we would need to add then is:

```
<set name="style" value="background: green; font-size: 10px; font-weight: bold; color: white;" ></set>
```

This way it would always look the same when you used the "saveButton" component.

So lets stitch this together as a whole. Since we are not adding any more components to our page, we can use the form as basis for it. Our completed component then becomes:

```
    <component jsfid="personregpanel" extends="form"
        id="personregpanel">
        <element jsfid="namepanel" renderId="1"></element>
        <element jsfid="t:htmlTag" renderId="2" id="pbr">
            <attributes>
                <set name="value" value="br" />
            </attributes>
        </element>
        <element jsfid="adresspanel" renderId="3">
            <symbols>
                <set name="adresslegend" value="#{messages['adresspanel.text']}"></set>
                <set name="streetadressLabel" value="#{messages['streetadress.label']}"></set>
                <set name="streetadressField" value="#{@managed-bean-name.person['streetadress']}"></set>
                <set name="zipcodeLabel" value="#{messages['zipcode.label']}"></set>
                <set name="zipcodeField" value="#{@managed-bean-name.person.post['zipcode']}"></set>
                <set name="cityLabel" value="#{messages['city.label']}"></set>
                <set name="cityField" value="#{@managed-bean-name.person.post['city']}"></set>
            </symbols>
        </element>
        <element jsfid="t:htmlTag" renderId="4" id="pbr2">
            <attributes>
                <set name="value" value="br" />
            </attributes>
        </element>
        <element jsfid="saveButton" renderId="5">
            <symbols>
                <set name="method" value="save"></set>
            </symbols>
        </element>
    </component>
```

Here we have defined our reusable components as elements (respecting the rendered) and defined actual values to substitute the symbols with.

Next we need to add an entry in the faces-config.xml file so that the navigation rules reflect our new page:

```
        <navigation-case>
                <from-outcome>personreg</from-outcome>
                <to-view-id>/personreg.jsf</to-view-id>
        </navigation-case>
```

Then we need to define the page as a view in the clay-views-config.xml file:

```
        <component jsfid="/personreg.jsf" extends="baseLayout">
                <symbols>
                        <set name="title" value="#{messages['personreg.title']}" />
                        <set name="bodyContent" value="/pages/personreg.html" />
                </symbols>
        </component>
```

The last thing we need to do is to add another entry in our menu structure which is defined in defaultLeftNav.html (in folder pages):

```
<li>
  <a jsfid="commandLink" action="personreg" allowBody="true" immediate="true"><span jsfid="outputText" value="#
{messages['menu.personreg']}" allowBody="false">Personreg</span></a>
</li>
```

Package, deploy and start the application. Then choose Personreg from the menu. You should now be presented with this:

ShaleClay10.png!

Enter some values for the different fields and push "Save". Choose one of the other pages, and the Personreg again. The data are still there because we declared our Person bean to have session scope.

To refine this you can substitute the zipcode field with a combobox that gets its values from the Post beans getZipcodes method (This is shown in the downloadable finished application). The HTML definition for it is:

```
        <select value="@zipcodeField" allowBody="true">
            <option value="@zipcodesList" />
        </select>
```

In addition we also have to declare what the symbol "zipcodesList" is to be replaced with when we use it on our page.

The values for such a list must be one of the following types:
SelectItem `SelectItem[]`, Collection or Map. We use SelectItem, so you need to add the following method to the Post bean:

```
    private void initZipcodesList() {
        zipcodeliste=new SelectItem[zipcodes.length];
        SelectItem selectItem;
        for(int i=0; i < zipcodes.length; i++)
        {
            selectItem=new SelectItem();
            selectItem.setLabel(zipcodes[i].toString());
            selectItem.setValue(zipcodes[i]);
            zipcodesliste[i]=selectItem;
        }

    }
```

And add a default constructor to it:

```
    public Post()
    {
        initZipcodesList();
    }
```

When you run this it will look like this:

ShaleClay11.png!

The Newscomponent and other components will follow shortly

Hermod Opstvedt February 2007