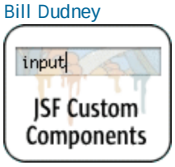
Creating JSF Custom Components

July 16, 2004

Bill Dudney



This article illustrates how to build custom components for use in web applications based on JavaServer Faces (JSF). While JSF comes with a standard set of components, one of the most-publicized features is the easy addition of new components. In this article, you will see just how easy it is to create new components that are fully functional and integrated into your web applications. Specifically, this article will show how to develop a component that allows users to enter valid credit card numbers.

Contents

A Very Quick JSF Introduction

At a very high level, JSF is an implementation of the Model 2 pattern, also referred to as Model-View-Controller (MVC), for the Web. The basic idea behind the pattern is to separate the concerns of your application; namely, the business logic, the user interface that allows manipulation business data, and the glue that makes the two able to communicate. Another way to think about MVC is that you don't want to put the discount calculation into the View, because that calculation is business logic. Instead, keep all of the business logic together in coherent abstractions referred to as the Model. Then, keep the complexity of your user interface together in coherent abstractions referred to as the View. Finally have the model and view talk together with a layer of "glue" called the Controller.

Components in JSF are part of the View layer. The view is built by composing groups of components into trees that together make up the user interface. The components themselves are fairly fine-grained and are typically reusable from one user interface to another. An example would be the `HtmlInputText` component that is part of the JSF standard. This component provides a text field for users to type in data. A group of inputs is composed into a view, and that view allows for manipulation of a particular piece of business information through the controller that sits between the view and model objects being edited. You can find a lot more good information at the official JSF site.

Components in JSF

Now that we have presented a high level view of what JSF is, let's dive into the particulars of what a component is. In JSF, a component is a group of interacting classes that together provide a reusable piece of web-based user interface code. A component is made up of three classes that work closely together.

First is the renderer, which creates the client-side representation of the component and takes any input from the client and transforms it into something the component can understand. The typical renderer implementation will generate HTML and understand how to transform values from HTML form POSTs into values the component understands. However this is not the only client type that can be rendered by a JSF renderer; for example, the reference implementation comes with a set of XUL renderers.

Second is the `UIComponent` subclass. This class is responsible for the data and behavior of the component on the server side; in other words, anything that the component is responsible for that is not related to rendering is found in the component class.

Finally, most JSF components will have a JSP custom action. The main functions of this class are to allow for configuration of the component in a JSP and to attach a particular renderer to the component. As an example to make

this clearer, consider the **UICommand** component (one of the standard components included in JSF). This component can be rendered as a link or a button. The JSP custom action is what makes the association -- for example, `h:commandButton` will render a button and `h:commandLink` will render an HTML link.

While we are discussing JSPs and JSF, I'd like to address one common misconception with JSF. JSPs are not the only way to compose a JSF user interface. In fact, it is fairly straightforward to build a JSF user interface in a servlet or in a plain Java class. JSPs were chosen as the "default" way to build JSF interfaces because of the broad knowledge base that exists in building web applications with JSPs.

Let's get out of the abstract and into the concrete. One of the standard **UIComponents** in JSF is the **HtmlInputText**. It is rendered by the **TextRenderer** (a concrete renderer in the reference implementation), and the final piece of the component is the `h:inputText` custom JSP action. These three classes together make up the standard component for users to type in small amounts of text. The tag makes the association between the component and its renderer. Figure 1 depicts the relationship between the three pieces of this component.

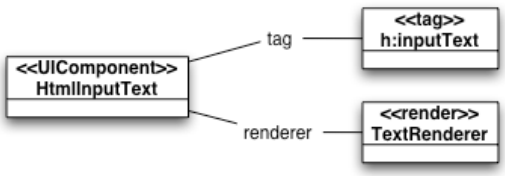


Figure 1. ***HtmlInputText*** *component*

Credit Card Input Component

Now that we are done with the basics of what a component is, let's go over the example custom component that will be fleshed out in this article. The example component allows users to input credit card numbers. A credit card number is defined to be 16 digits with or without dashes every four digits. The component will validate that the values entered match this definition. In order to implement this component, we will create four classes, the **UIComponent** subclass, the renderer, the JSP custom action, and a validator to ensure that any entered value matches this definition of a credit card number. Figure 2 shows the classes and their relationships.

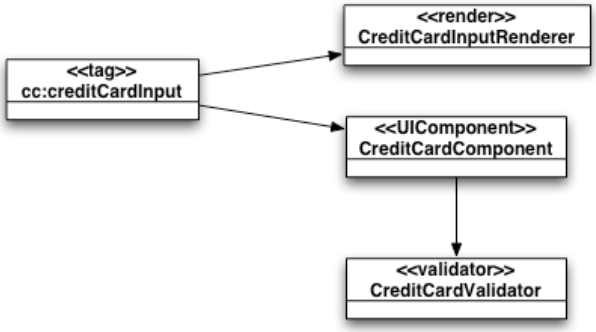


Figure 2. Credit card component

The first class we will look at is the component. Since the credit card component allows users to input text values, we will subclass **UIInput** to make the component. Here is the code:

```
package example.component;
public class UICreditCardInput extends UIInput {
  public static final String
         CREDIT_CARD_FAMILY = "CCFAMILY";

  public UICreditCardInput() {
    super();
    addValidator(new CreditCardValidator());
  }

  public String getFamily() {
    return CREDIT_CARD_FAMILY;
  }
}
```

Notice how simple this component is in terms of code content. Many custom components will be as simple as this; of course, there are many that will be more complex. And, no surprise, the complexity of the code depends directly on the complexity of the component being written. As an example, the calendar component in the MyFaces project is about 125

lines of generated code. This component is so simple because it is not adding any additional data fields to **UIInput**, so it's able to reuse most of the functionality of its superclass.

In order to accomplish the validation specified in the requirements for the credit card component, we add a validator in the constructor. Next is the `getFamily` method. This method simply returns the *family* of the component. This family (and thus the method that retrieves it) is significant, because this value is used to look up the renderer when it comes time to make HTML. We will see more on how the render kit uses the family to find a particular renderer later when we discuss rendering.

Next up is the validator that makes sure the credit card number fits the criteria (16 digits, with or without a dash every four characters). This is a very straightforward task with the regexp abilities built into J2SE 1.4 and later. Here is the code:

```
public class CreditCardValidator implements
    Validator {
...
  public void validate(FacesContext context,
      UIComponent component, Object value)
      throws ValidatorException {
    if (null != value) {
      if (!(value instanceof String)) {
        throw new IllegalArgumentException(
            "The value must be a String");
      }
      String ccNum = (String) value;
      // do some asserts that value matches a
      // regex...
      Pattern withDashesPattern = Pattern
          .compile("\\d\\d\\d\\d-\\d\\d\\d\\d-"
              + "\\d\\d\\d\\d-\\d\\d\\d\\d");
      Pattern noDashesPattern = Pattern
          .compile("\\d\\d\\d\\d\\d\\d\\d\\d\\d"
              + "\\d\\d\\d\\d\\d\\d\\d\\d");
      Matcher withDashesMatcher = withDashesPattern
          .matcher(ccNum);
      Matcher noDashesMatcher = noDashesPattern
          .matcher(ccNum);
      if (!withDashesMatcher.matches()
          && !noDashesMatcher.matches()) {
        throw new ValidatorException(
            new FacesMessage(
                "The Credit Card Number must "
                    + "have 16 digits with or "
                    + "without dashes every "
                    + "four digits"));
      }
    }
  }
...
}
```

The important thing to note here is the response to error; if the string does not match one or the other of the **Matcher**s, then a new **FacesMessage** is created and a **ValidatorException** is thrown. JSF will interpret the **ValidationException** and get the message from it so that you can display it in your faces interface.

While it's technically not part of the component, I suspect that the ability to have a validator connected to a component by default will be one of the reasons people will develop custom components. There is a lot more info on validation in the spec.

The next part of the component to consider is the renderer. Keep in mind that the renderer is responsible for turning the component into HTML and taking posted HTML and turning it into values suitable for the component. (This process is known as *rendering* or as *encoding and decoding*.) In other words, the renderer is responsible for building the HTML that is used in the UI and taking any form posts and passing the values in the post back to the component. Here is the code for the **CreditCard** renderer.

```
public class CreditCardInputRenderer extends
    Renderer {
...
  public void decode(FacesContext context,
      UIComponent component) {
    assertValidInput(context, component);

    if (component instanceof UIInput) {
      UIInput input = (UIInput) component;
      String clientId = input
          .getClientId(context);
```

```
      Map requestMap = context
          .getExternalContext()
          .getRequestParameterMap();
      String newValue = (String) requestMap
          .get(clientId);
      if (null != newValue) {
        input.setSubmittedValue(newValue);
      }
    }
  }

  public void encodeEnd(FacesContext ctx,
      UIComponent component) throws IOException {
    assertValidInput(ctx, component);
    ResponseWriter writer = ctx
        .getResponseWriter();
    writer.startElement("input", component);
    writer.writeAttribute("type", "text", "text");
    String id = (String) component
        .getClientId(ctx);
    writer.writeAttribute("id", id, "id");
    writer.writeAttribute("name", id, "id");
    String size = (String) component
        .getAttributes().get("size");
    if (null != size) {
      writer.writeAttribute("size", size, "size");
    }
    Object currentValue = getValue(component);
    writer.writeAttribute("value",
        formatValue(currentValue), "value");
    writer.endElement("input");
  }

  protected Object getValue(UIComponent component) {
    Object value = null;
    if (component instanceof UIInput) {
      value = ((UIInput) component)
          .getSubmittedValue();
    }
    // if its not a UIInput or the submitted value
    // was null then get the value (it should
    // always be a UIInput)
    if (null == value
        && component instanceof ValueHolder) {
      value = ((ValueHolder) component)
          .getValue();
    }

    return value;
  }

  private String formatValue(Object currentValue) {
    // this should be a bit more sophisticated
    // in essence what should happen here is any
    // conversion that needs to take place.
    return currentValue.toString();
  }

  private void assertValidInput(
      FacesContext context, UIComponent component) {
    if (context == null) {
      throw new NullPointerException(
          "context should not be null");
    } else if (component == null) {
      throw new NullPointerException(
          "component should not be null");
    }
  }

}
```

This class has the bulk of the complexity for our example. This class is so complex because the particular renderers (text, button, etc.) are not publicly part of the specification and thus we can't subclass them directly. I don't think that is likely to change because rendering is one place in which commercial vendors can compete. Enough about politics; let's look at what is this class actually doing.

The first method is `decode(FacesContext, UIComponent)`. This method is responsible for taking any parameters that were passed in from a form post and setting the value on the component. The first thing the method does is to assert that the context and component parameters are not null. This is part of the specification and should be done in every method of similar signature in any renderer. Next, the method avoids anything but **`UIInput`** components. If the

component is a **UIInput**, then any new value is extracted from the request and put on the component as the submittedValue. Notice that no attempt to validate the value is made here in the renderer. Validation is handled separately by JSF during either the *Process Validations* or the *Apply Request Values* phase depending on the immediate property (there is a lot more detail on validation in section 3.5 of the final JSF specification).

The next method is encodeEnd(**FacesContext**, **UIComponent**). This method is responsible for building the HTML to represent the component in the browser. There are actually three encoding methods: encodeBegin(**FacesContext**, **UIComponent**), encodeChildren(**FacesContext**, **UIComponent**), and the one implemented here, encodeEnd(**FacesContext**, **UIComponent**). Since the **UICreditCardInput** component is fairly simple and does not have children, there is no need to implement the first two methods. For more complex components that render their child components, you would have the encodeBegin method start the element for the root component (the one being encoded), the encodeChildren would cause all of the children to be encoded, and finally, the encodeEnd method would close the element.

The first thing this method does is to assert that the parameters are valid. Next, this method uses the *element*-oriented methods on the writer that are part of the specification. This method of building the HTML is fine for very simple components like this, but for anything more complex the encode*XXX* methods end up full of HTML-oriented strings. HTML strings embedded in your code are never a good idea, and make maintenance a headache. In the downloadable code, I offer an alternative implementation that uses JDom. The important thing for you to know about this method is that you are free to build the HTML in any way you want, in whatever technology you are comfortable with, as long as it eventually makes it into a stream of characters and is written into the **ResponseWriter**.

The last Java class we have to look at is the JSP action, or tag. This class is responsible for creating the component (handled by the **UIComponentTag** class from which you will typically subclass), attaching the renderer to the component, and finally, setting the fields on the components based on the values supplied in the JSP. Here is the code for the **CreditCardInputTag**.

```
public class CreditCardInputTag extends
    UIComponentTag {

  private static final String CCI_COMP_TYPE =
    "CREDIT_CARD_INPUT";

  private static final String CCI_RENDER_TYPE =
    "CREDIT_CARD_RENDERER";
...
  public String getComponentType() {
    return CCI_COMP_TYPE;
  }

  public String getRendererType() {
    return CCI_RENDER_TYPE;
  }
...
  protected void setProperties(
      UIComponent component) {
    FacesContext context = FacesContext
        .getCurrentInstance();
    super.setProperties(component);
    if (null != size) {
      component.getAttributes().put("size", size);
    }
    if (null != value) {
      if (isValueReference(value)) {
        ValueBinding vb = context
            .getApplication().createValueBinding(
                value);
        component.setValueBinding("value", vb);
      } else {
        ((UIInput) component).setValue(value);
      }
    }
  }
}
```

Apart from the accessor methods (available in the download) there are three methods to look at. The first is getComponentType(). This method returns the type of the component that should be created by the tag (the type is sent to a factory for resolution to a class name and then an instance is created). The next method, getRendererType(), returns the type of the renderer. (The same factory story: the type is sent to a factory and the class name is matched, and a new instance is returned if needed. We will discuss this later in the discussion of the configuration file.) These two methods are used to make the connection between the renderer and the component.

This is a very powerful tool that we can use to associate different renderers with components. For example, let's consider one of the standard components, **UICommand**. This component can be rendered as an HTML link or as a button.

The association is made via the JSP with the use of two different tags. To render the command as a button, we use the `h:commandButton` tag; to render as a link, we use the `h:commandLink` tag.

Next let's take a quick look at a very simple JSP that we can use to see the component in action.

```
<%@taglib prefix="h"
  uri="http://java.sun.com/jsf/html"%>
<%@taglib prefix="f"
    uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="cc"
    uri="http://bill.dudney.net/cc/component"%>
<html>
  <head>
    <title>Credit Card Component Example</title>
  </head>
  <body>
   <h1>Hello and welcome to the example</h1>
   <f:view>
   <p>
   <h:messages id="messageList" showSummary="true"/>
   </p>
   <h:form>
     <h:outputLabel for="test">
       <h:outputText id="testLabel" value="Credit Card Number:"/>
     </h:outputLabel>
     <cc:creditCardInput id="test" size="19"
                         value="#{page.ccNum}"/>
     <br/>
     <h:commandButton value="Save"
                      action="#{page.saveNumber}"/>
   </h:form>
   </f:view>
  </body>
</html>
```

This is all very standard JSP stuff: the **`CreditCard`** tag lib is imported with the `taglib` directive (along with the required JSF tags). The **`UICreditCardInput`** is created by the `cc:creditCardInput` tag, as we saw earlier in the code.

Figure 3 shows the component in action.

Figure 3. The credit card component rendered in a browser

Notice the error text in Figure 3. The user typed the character `a` into the credit card number (only numbers are allowed) and the validator rejected the input and supplied an error message.

Finally, let's take a quick look at the configuration file that tells JSF about the component and its renderer. Here is the code for the *faces-config.xml* file used in this example to configure the **`UICreditCardInput`** component and its renderer.

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
  <component>
      <component-type>
```

```
        CREDIT_CARD_INPUT
      </component-type>
        <component-class>
          example.component.UICreditCardInput
        </component-class>
  </component>

  <render-kit>

    <renderer>
      <description>
        Renderer for the credit card component.
      </description>
      <component-family>CCFAMILY</component-family>
      <renderer-type>
        CREDIT_CARD_RENDERER
      </renderer-type>
      <renderer-class>
        example.renderer.CreditCardInputRenderer
      </renderer-class>
    </renderer>
  </render-kit>
</faces-config>
```

The `component` tag tells JSF the component type and the class. This is where the association is made between the type and the class of the component (that was alluded to in the earlier discussion of the **`CreditCardInputTag`**). This configuration file will end up being *META-INF/faces-config.xml* in the .jar file that represents this component. JSF will look for such a file name in each of the .jar files that are loaded at runtime (in the *WEB-INF/lib* directory for .war files) and use each of them in its configuration. In this way, multiple component .jar files can be combined into one web application, and all of the components described in each .jar will be available to the application.

Conclusion

This component is built from three parts: the **`UIComponent`** subclass, the renderer, and the JSP custom action. We have seen the implementation of all three parts as well as the JSP that puts them all together. All told, the credit card input component and example amount to only a few dozen lines of code. See how easy that is? Of course, as your components increase in complexity, the amount of code will increase, as well. The beauty of the JSF model is that the additional code is related to the complexity of the component and not JSF.

Good luck building your own components! You can download the code for this article. I hope you enjoy building your own components.

Bill Dudney is an architect with Object Systems Group. He has been doing distributed computing for 14 years starting way back at NASA building software to manage the mass properties of the Space Shuttle.

Related Topics >> Web Services and XML |

Article Links >> Login or register to post comments Printer-friendly version ShareThis 42606 reads

Comments

The link is broken, you can

by victorbr - 2009-12-23 09:33

The link is broken, you can send to me the source code to my email (victorbaro3@hotmail.com) or repair the link, thanks

Login or register to post comments

The link to source code is

by rtarasov - 2010-09-19 03:40

The link to source code is broken. Could you please send the source code to rtarasov@gmail.com. Thanks in advance!

Login or register to post comments

Creating JSF Custom

by vinodeuro - 2010-10-10 22:57

This article is very useful but your downloading link is broken. so could you please send me source code on my following email-id: vinod.pokharkar@gmail.com Thanks Vinod

Login or register to post comments