JSF 2 fu, Part 3: Event handling, JavaScript, and Ajax

Enhance composite components using more new JSF 2 features

Skill Level: Intermediate

David Geary
President
Clarity Training, Inc.

14 Jul 2009

Java™Server Faces (JSF) 2 Expert Group member David Geary wraps up his three-part series on JSF 2's new features. Find out how to use the framework's new event model and built-in support for Ajax to make your reusable components all the more powerful.

One of JSF's biggest selling points is that it is a component-based framework. That means you can implement components that you, or other people, can reuse. That powerful reuse mechanism was, for the most part, rendered inconsequential in JSF 1 because it was so difficult to implement components.

As you saw in Part 2, however, JSF 2 makes it easy to implement components — with no Java code and no configuration — with a new feature called *composite components*. That feature may very well be the most important part of JSF 2, because it finally realizes the potential of JSF components.

In this third and final article on JSF 2, I will show you how to build on the composite-component feature by using the new Ajax and event-handling capabilities also introduced in JSF 2, with the following tips for getting the most out of JSF 2:

- Tip 1: Componentize
- Tip 2: Ajaxify

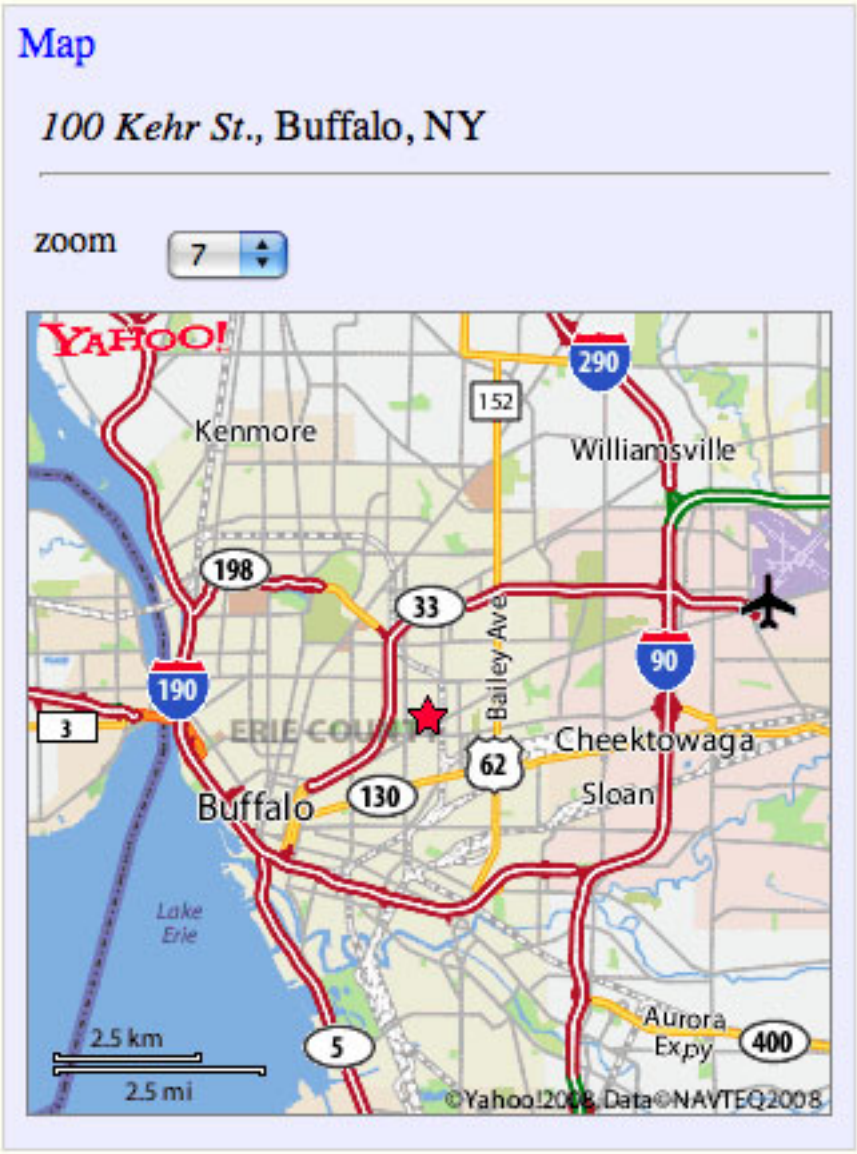
- Tip 3: Show progress

In the first tip, I'll briefly review two components that I discussed at length in Part 2. In the subsequent tips, I'll show you how to transform those components using Ajax and event-handling.

Tip 1: Componentize

The places application, which I introduced in Part 1, contains a number of composite components. One is the `map` component, which displays a map of an address, complete with a zoom-level pull-down menu, as shown in Figure 1:

Figure 1. The places application's map component



A truncated listing of the `map` component is shown in Listing 1:

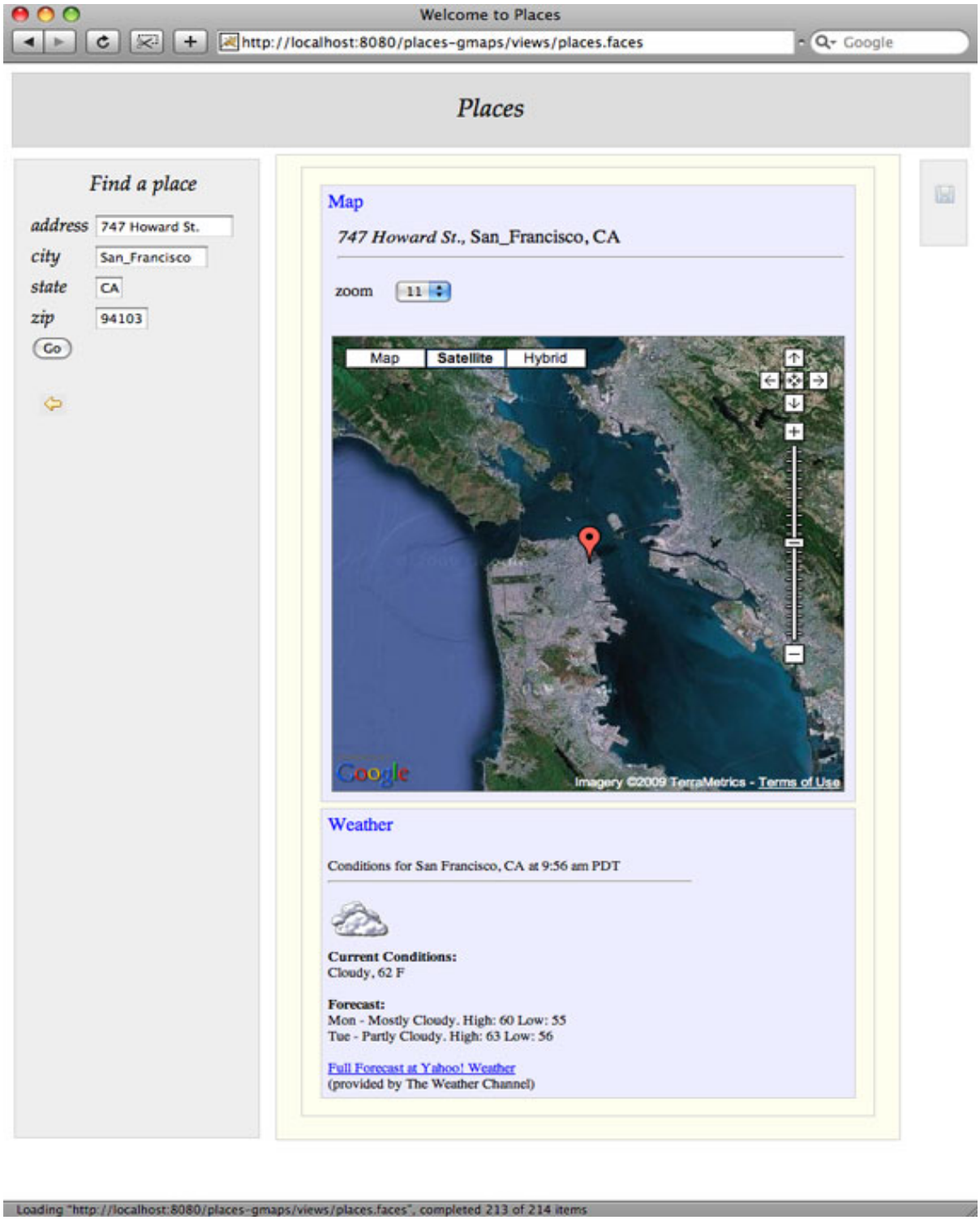
Listing 1. The map component

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<html xmlns="http://www.w3.org/1999/xhtml"
    ...
    xmlns:composite="http://java.sun.com/jsf/composite"
    xmlns:places="http://java.sun.com/jsf/composite/components/places">

   <!-- INTERFACE -->
   <composite:interface>
     <composite:attribute name="title"/>
   </composite:interface>

   <!-- IMPLEMENTATION -->
   <composite:implementation">
     <div class="map">
```

```
        ...
        <h:panelGrid...>
          <h:panelGrid...>
             <h:selectOneMenu onchange="submit()"
                  value="#{cc.parent.attrs.location.zoomIndex}"
                  valueChangeListener="#{cc.parent.attrs.location.zoomChanged}"
                  style="font-size:13px;font-family:Palatino">

               <f:selectItems value="#{places.zoomLevelItems}"/>

             </h:selectOneMenu>
           </h:panelGrid>
         </h:panelGrid>

         <h:graphicImage url="#{cc.parent.attrs.location.mapUrl}"
           style="border: thin solid gray"/>
        ...

      </div>
    ...

  </composite:implementation>
</html>
```

One of the great things about components is that you can replace them with higher-powered alternatives without disturbing any of the surrounding functionality. For example, in Figure 2, I've replaced the `image` component from Listing 1 with a Google Maps component, courtesy of GMaps4JSF (see Resources):

Figure 2. GMaps4JSF's map image



The updated (and truncated) code for the `map` component is shown in Listing 2:

Listing 2. Replacing the map image with a GMaps4JSF component

```
<h:selectOneMenu onchange="submit()"
              value="#{cc.parent.attrs.location.zoomIndex}"
              valueChangeListener="#{cc.parent.attrs.location.zoomChanged}"
              style="font-size:13px;font-family:Palatino">

  <f:selectItems value="#{places.zoomLevelItems}"/>

</h:selectOneMenu>

...

<m:map id="map" width="420px" height="400px"
     address="#{cc.parent.attrs.location.streetAddress}, ..."
     zoom="#{cc.parent.attrs.location.zoomIndex}"
     renderOnWindowLoad="false">

  <m:mapControl id="smallMapCtrl"
              name="GLargeMapControl"
          position="G_ANCHOR_TOP_RIGHT"/>

  <m:mapControl  id="smallMapTypeCtrl" name="GMapTypeControl"/>
  <m:marker id="placeMapMarker"/>

</m:map>
```

To use a GMaps4JSF component, I replaced the `<h:graphicImage>` tag with an `<m:map>` tag from the GMaps4JSF component set. It was also simple to hook the GMaps4JSF component up to the zoom pull-down, simply by specifying the correct backing-bean property for the `<m:map>` tag's `zoom` attribute.

Speaking of zoom levels, notice that when a user changes the zoom level, I force a form submit with the `<h:selectOneMenu>`'s `onchange` attribute, as shown in the first partially boldfaced line in Listing 1. That form submission triggers the JSF life cycle, which ultimately pushes the new value for the zoom level into the `zoomIndex` property of a `location` bean stored in the parent composite component. That bean property is bound to the input component, in the first line of Listing 2.

Because I didn't specify any navigation for the form submit associated with zoom-level change, JSF refreshes the same page after handling the request, redrawing the map image to reflect the new zoom level. However, that page refresh also redraws the entire page even though the only change is in the map image. In Tip 2: Ajaxify, I'll show you how to use Ajax to redraw only the image in response to a zoom-level change.

The login component

Another component used in the places application is the `login` component. Figure 3 shows the login component in action:

Figure 3. The login component

Listing 3 shows the markup that created the `login` component shown in Figure 3:

Listing 3. Minimalist login: Just the required attributes

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:util="http://java.sun.com/jsf/composite/components/util">

  <util:login loginAction="#{user.login}"
              managedBean="#{user}"/>

</ui:composition>
```

The `login` component has just two required attributes:

- `loginAction`: A login action method
- `managedBean`: A managed bean with name and password properties

The managed bean specified in Listing 3 is shown in Listing 4:

Listing 4. User.groovy

```
package com.clarity

import javax.faces.context.FacesContext
import javax.faces.bean.ManagedBean
import javax.faces.bean.SessionScoped

@ManagedBean()
@SessionScoped

public class User {
  private final String VALID_NAME     = "Hiro"
  private final String VALID_PASSWORD = "jsf"

  private String name, password;

  public String getName() { name }
  public void setName(String newValue) { name = newValue }

  public String getPassword() { return password }
  public void setPassword(String newValue) { password = newValue }
```

```
  public String login() {
    "/views/places"
  }

  public String logout() {
    name = password = nameError = null
    "/views/login"
  }
}
```

The managed bean in Listing 4 is a Groovy bean. Using Groovy instead of the Java language doesn't buy me much in this case, other than freeing me from the drudgery of semicolons and return statements. However, in Tip 2's Validation section, I'll show you a more compelling reason to use Groovy for the `User` managed bean.

Most of the time you'll want to configure login components fully with prompts and button text, as shown in Figure 4:

Figure 4. A fully configured login component

Listing 5 shows the markup that generated the `login` component in Figure 4:

Listing 5. Configuring the login component

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:util="http://java.sun.com/jsf/composite/components/util">

  <util:login loginPrompt="#{msgs.loginPrompt}"
              namePrompt="#{msgs.namePrompt}"
          passwordPrompt="#{msgs.passwordPrompt}"
         loginButtonText="#{msgs.loginButtonText}"
             loginAction="#{user.login}"
             managedBean="#{user}"/>

</ui:composition>
```

In Listing 5, I obtain strings for prompts and the login button text from a resource

bundle.

Listing 6 defines the `login` component:

Listing 6. Defining the login component

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Usage:

  <util:login loginPrompt="#{msgs.loginPrompt}"
               namePrompt="#{msgs.namePrompt}"
           passwordPrompt="#{msgs.passwordPrompt}"
          loginButtonText="#{msgs.loginButtonText}"
              loginAction="#{user.login}"
              managedBean="#{user}">

    <f:actionListener for="loginButton"
                      type="com.clarity.LoginActionListener"/>

  </util:login>

  managedBean must have two properties: name and password.

  The loginAction attribute must be an action method that takes no
  arguments and returns a string. That string is used to navigate
  to the page the user sees after logging in.

  This component's loginButton is accessible so that you can
  add action listeners to it, as depicted above. The class specified
  in f:actionListener's type attribute must implement the
  javax.faces.event.ActionListener interface.

 -->

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:composite="http://java.sun.com/jsf/composite">

  <!-- INTERFACE -->
  <composite:interface>

    <!-- PROMPTS -->
    <composite:attribute name="loginPrompt"/>
    <composite:attribute name="namePrompt"/>
    <composite:attribute name="passwordPrompt"/>

    <!--  LOGIN BUTTON -->
    <composite:attribute name="loginButtonText"/>

    <!-- loginAction is called when the form is submitted -->
    <composite:attribute name="loginAction"
             method-signature="java.lang.String login()"
                      required="true"/>

    <!-- You can add listeners to this actionSource:  -->
    <composite:actionSource name="loginButton" targets="form:loginButton"/>

    <!-- BACKING BEAN -->
    <composite:attribute name="managedBean" required="true"/>
  </composite:interface>

  <!-- IMPLEMENTATION -->
```

```
  <composite:implementation>
    <div class="prompt">
      #{cc.attrs.loginPrompt}
    </div>

    <!-- FORM -->
    <h:form id="form">
      <h:panelGrid columns="2">

        <!-- NAME AND PASSWORD FIELDS -->
        #{cc.attrs.namePrompt}
        <h:inputText id="name"
                  value="#{cc.attrs.managedBean.name}"/>

        #{cc.attrs.passwordPrompt}
        <h:inputSecret id="password" size="8"
          value="#{cc.attrs.managedBean.password}"/>

      </h:panelGrid>

      <p>
        <!-- LOGIN BUTTON -->
        <h:commandButton id="loginButton"
          value="#{cc.attrs.loginButtonText}"
          action="#{cc.attrs.loginAction}"/>
      </p>
    </h:form>
  </composite:implementation>
</html>
```

Like the `map` component, the `login` component could use an Ajax upgrade. In the next tip, under Validation, I'll show you how to add Ajax validation to the login component.

Tip 2: Ajaxify

Ajax typically requires two steps that are normally not done for non-Ajax HTTP requests: partial processing of forms on the server, and a subsequent partial rendering of the Document Object Model (DOM) on the client.

Partial processing and rendering

JSF 2 supports partial processing and partial rendering by splitting the JSF life cycle into two distinct logical portions: execute and render. Figure 5 highlights the execute portion:

Figure 5. The execute portion of the JSF life cycle

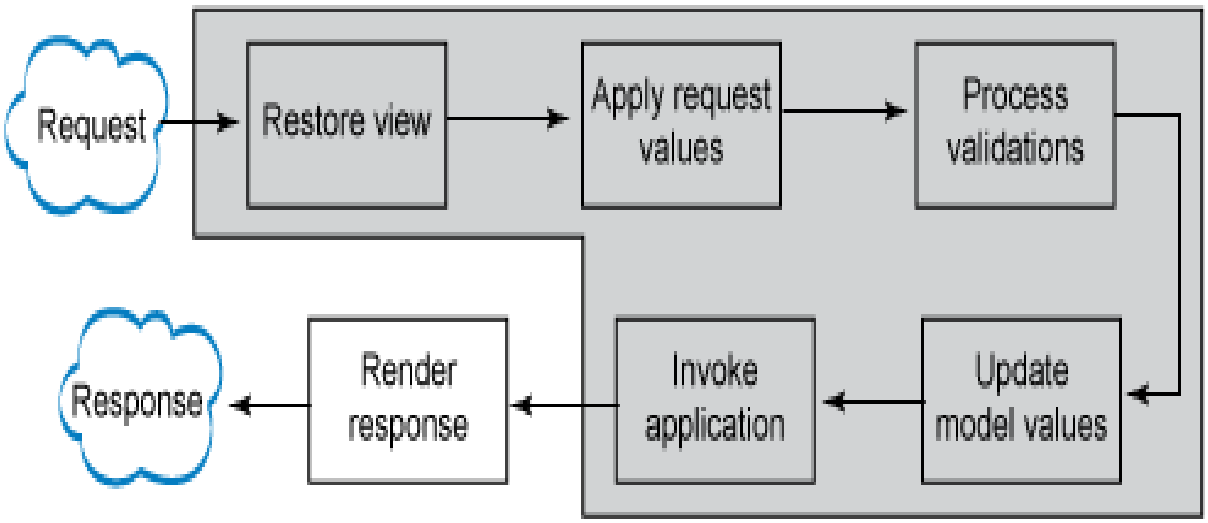
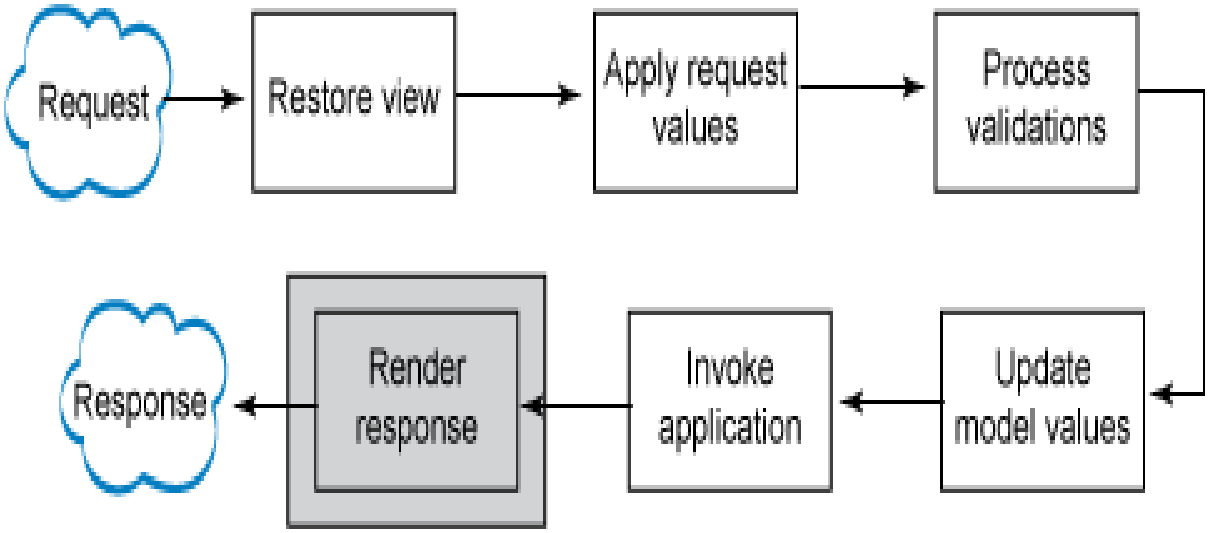


Figure 6 highlights the JSF life cycle's render portion:

Figure 6. The render portion of the JSF life cycle



The idea behind the life cycle's execute and render portions is simple: you can specify components that JSF executes (processes) on the server, and components that JSF renders when an Ajax call returns. You do that with `<f:ajax>`, which is new for JSF 2, as shown in Listing 7:

Listing 7. An Ajax zoom menu

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<h:selectOneMenu id="menu"
     value="#{cc.parent.attrs.location.zoomIndex}"
     style="font-size:13px;font-family:Palatino">

  <f:ajax event="change" execute="@this" render="map"/>
  <f:selectItems value="#{places.zoomLevelItems}"/>

</h:selectOneMenu>
```

```
<m:map id="map"...>
```

Listing 7 is a modification of the menu shown in the first line of Listing 2: I've removed the `onchange` attribute from Listing 2 and added an `<f:ajax>` tag. That `<f:ajax>` tag specifies:

- The event that triggers the Ajax call
- A component to execute on the server
- A component to render on the client

When the user selects an item from the zoom menu, JSF makes an Ajax call to the server. Subsequently, JSF passes the menu through the life cycle's execute portion (`@this` signifies `<f:ajax>`'s surrounding component), and updates the menu's `zoomIndex` during the Update Model Values phase of the life cycle. When the Ajax call returns, JSF renders the map component, which uses the (newly set) zoom index to redraw the map, and now you have an Ajaxified zoom menu with the addition of one line of XHTML.

But things can get simpler still, because JSF provides default values for the `event` and `execute` attributes.

Each JSF component has a default event that triggers Ajax calls if you embed an `<f:ajax>` tag inside the component tag. For menus, that event is the `change` event. That means I can get rid of the `<f:ajax>`'s `event` attribute in Listing 7. The default for `<f:ajax>`'s `execute` attribute is `@this`, which signifies `<f:ajax>`'s surrounding component. In this example, that component is the menu, so I can also get rid of the `execute` attribute.

By using default attribute values for `<f:ajax>`, I can reduce Listing 7 to Listing 8:

Listing 8. Simpler version of an Ajax zoom menu

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<h:selectOneMenu id="menu"
     value="#{cc.parent.attrs.location.zoomIndex}"
     style="font-size:13px;font-family:Palatino">

  <f:ajax render="map"/>
  <f:selectItems value="#{places.zoomLevelItems}"/>

</h:selectOneMenu>

<m:map id="map"...>
```

That's how easy it is to add Ajax to your components with JSF 2. Of course, the preceding example is pretty simple: I am simply redrawing only the map instead of the entire page when the user selects a zoom level. Some operations, such as validating individual fields in a form, are more complicated, so next I'll tackle that use

case.

Validation

It's a good idea to validate fields, and provide immediate feedback, when a user tabs out of a field. For example, in Figure 7, I'm using Ajax to validate the name field:

Figure 7. Ajax validation

The markup for the name field is shown in Listing 9:

Listing 9. The name field

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<h:panelGrid columns="2">
  #{cc.attrs.namePrompt}
  <h:panelGroup>
    <h:inputText id="name" value="#{cc.attrs.managedBean.name}"
       valueChangeListener="#{cc.attrs.managedBean.validateName}">

       <f:ajax event="blur" render="nameError"/>

     </h:inputText>

     <h:outputText id="nameError"
       value="#{cc.attrs.managedBean.nameError}"
       style="color: red;font-style: italic;"/>
  </h:panelGroup>
  ...
</h:panelGrid>
```

Once again, I use `<f:ajax>`, only this time the default event for inputs — `change` — won't do, so I specify `blur` for the event that triggers the Ajax call. When the user tabs out of the name field, JSF makes an Ajax call to the server and runs the `name` input component through the execute portion of the life cycle. This means that JSF will invoke the `name` input's value-change listener specified in Listing 9 during the Process Validations phase of the life cycle. Listing 10 shows that value-change listener:

Listing 10. The validateName() method

```
package com.clarity

import javax.faces.context.FacesContext
import javax.faces.bean.ManagedBean
import javax.faces.bean.SessionScoped
import javax.faces.event.ValueChangeEvent
import javax.faces.component.UIInput

@ManagedBean()
@SessionScoped

public class User {
  private String name, password, nameError;

  ...

  public void validateName(ValueChangeEvent e) {
    UIInput nameInput = e.getComponent()
    String name = nameInput.getValue()

    if (name.contains("_"))   nameError = "Name cannot contain underscores"
    else if (name.equals("")) nameError = "Name cannot be blank"
    else                      nameError = ""
  }

  ...
}
```

The value-change listener — the `user` managed bean's `validateName()` method — validates the name field and updates the `user` managed bean's `nameError` property.

After the Ajax call returns, by virtue of the `render` attribute of the `<f:ajax>` tag in Listing 9, JSF renders the `nameError` output. That output shows the `user` managed bean's `nameError` property.

Multifield validation

In the preceding subsection, I showed you how to perform Ajax validation on a single field. Sometimes, though, you need to validate multiple fields at the same time. For example, Figure 8 shows the places application validating the name and password fields together:

Figure 8. Validating multiple fields

I validate the name and password fields together when the user submits the form, so I don't need Ajax for this example. Instead I will use JSF 2's new event system, as shown in Listing 11:

Listing 11. Using <f:event>

```
<h:form id="form" prependId="false">

  <f:event type="postValidate"
       listener="#{cc.attrs.managedBean.validate}"/>
  ...
</h:form>

<div class="error" style="padding-top:10px;">
  <h:messages layout="table"/>
</div>
```

In Listing 11, I use `<f:event>`, which — like `<f:ajax>` — is new for JSF 2. The `<f:event>` tag is similar to `<f:ajax>` in another respect, too: it's simple to use.

You put an `<f:event>` tag inside a component tag, and when the specified event (specified with the `type` attribute) occurs to that component, JSF invokes a method, specified with the `listener` attribute. So, in English, what the `<f:event>` tag in Listing 11 means is: After validating the form, invoke the `validate()` method on the managed bean that the user passed to this composite component. That method is shown in Listing 12:

Listing 12. The validate() method

```
package com.clarity

import javax.faces.context.FacesContext
import javax.faces.bean.ManagedBean
import javax.faces.bean.SessionScoped
import javax.faces.event.ValueChangeEvent
import javax.faces.component.UIInput
```

```
@ManagedBean()
@SessionScoped

public class User {
  private final String VALID_NAME     = "Hiro";
  private final String VALID_PASSWORD = "jsf";

  ...

  public void validate(ComponentSystemEvent e) {
    UIForm form = e.getComponent()
    UIInput nameInput = form.findComponent("name")
    UIInput pwdInput = form.findComponent("password")

    if ( ! (nameInput.getValue().equals(VALID_NAME) &&
        pwdInput.getValue().equals(VALID_PASSWORD))) {

      FacesContext fc = FacesContext.getCurrentInstance()
      fc.addMessage(form.getClientId(),
        new FacesMessage("Name and password are invalid. Please try again."))
      fc.renderResponse()
    }
  }

  ...
}
```

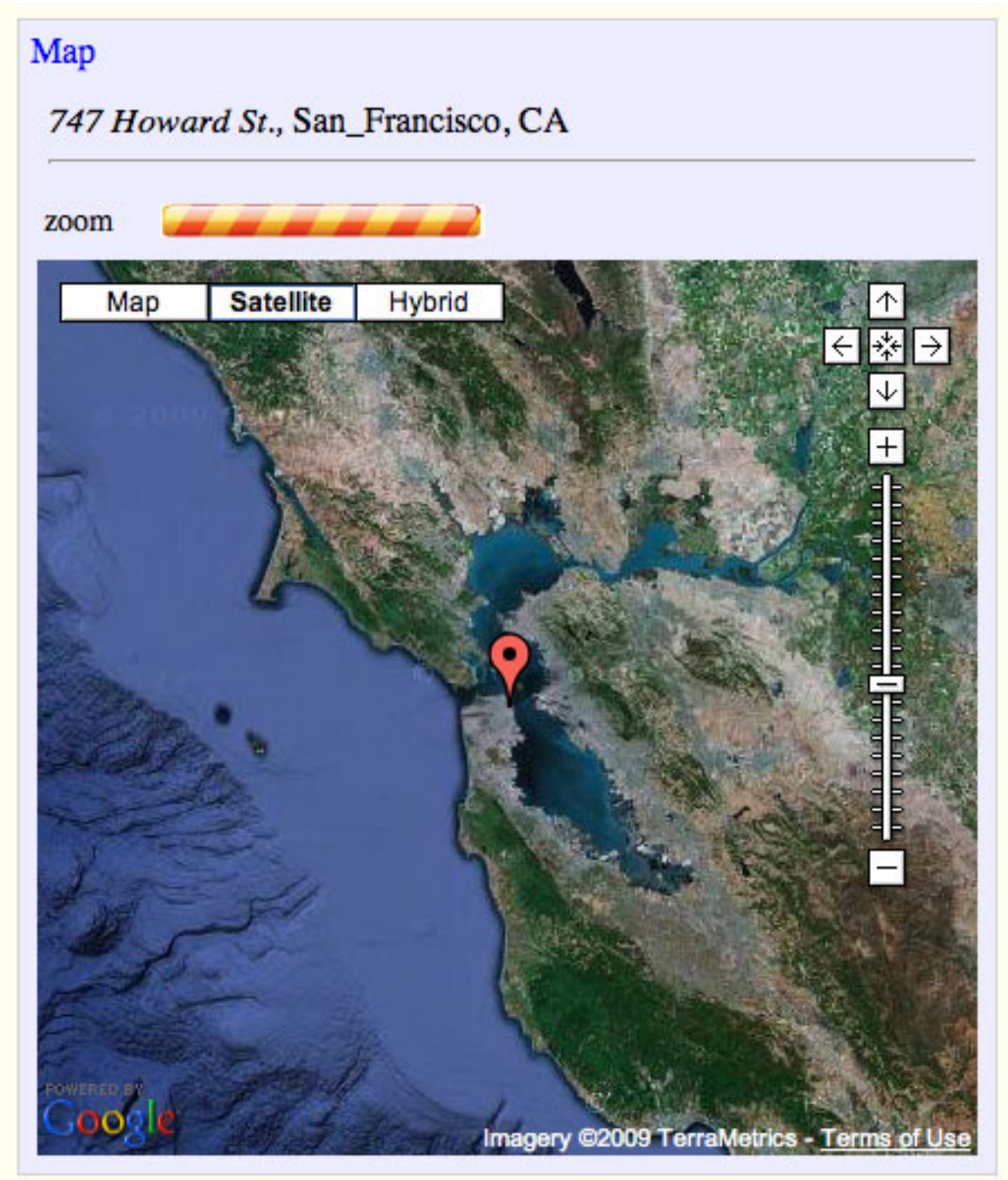
JSF passes the `validate()` method in Listing 12 a component system event, from which the method obtains a reference to the component the event applies to — the login form. From the form, I use the `findComponent()` method to get the name and password components. If those components' values are not Hiro and jsf, respectively, I store a message on the faces context and tell JSF to proceed immediately to the life cycle's Render Response phase. This way, I avoid the Update Model Values phase, which would push the bad name and password through to the model (see Figure 5).

You may have noticed that the validation methods in Listing 10 and Listing 12 are written in Groovy. Unlike Listing 4, where the only advantage to using Groovy was freedom from semicolons and return statements, the Groovy code in Listing 10 and Listing 12 frees me from casting. For example, in Listing 10, `ComponentSystemEvent.getComponent()` and `UIComponent.findComponent()` both return type `UIComponent`. With the Java language, I would have to cast the return values of those methods. Groovy does the casting for me.

Tip 3: Show progress

In Ajaxify, I showed you how to Ajaxify the zoom menu for the `map` component, so that the places application redraws only the map portion of the page when the user changes the zoom level. Another common Ajax use case is to provide some feedback to the user that an Ajax event is progressing, as shown in Figure 9:

Figure 9. A progress bar



In Figure 9, I've replaced the zoom menu with an animated GIF that displays while the Ajax call progresses. When the Ajax call is complete, I replace the progress indicator with the zoom menu. Listing 13 shows how it's done:

Listing 13. Monitoring an Ajax request

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<h:selectOneMenu id="menu"
     value="#{cc.parent.attrs.location.zoomIndex}"
     style="font-size:13px;font-family:Palatino">

  <f:ajax render="map" onevent="zoomChanging"/>
  <f:selectItems value="#{places.zoomLevelItems}"/>

  ...
</h:selectOneMenu>
...
<h:graphicImage id="progressbar" style="display: none"
              library="images" name="orange-barber-pole.gif"/>
```

In Listing 13, I add a progress-bar image, which is initially not displayed, and specify the `onevent` attribute for `<f:ajax>`. That attribute references a JavaScript function, shown in Listing 14, that JSF calls while the Ajax call initiated in Listing 13 progresses:

Listing 14. JavaScript that responds to an Ajax request

```
function zoomChanging(data) {
  var menuId = data.source.id;
  var progressbarId = menuId.substring(0, menuId.length - "menu".length)
      + "progressbar";

  if (data.name == "begin") {
    Element.hide(menuId);
    Element.show(progressbarId);
  }
  else if (data.name == "success") {
    Element.show(menuId);
    Element.hide(progressbarId);
  }
}
```

JSF passes the function in Listing 14 an object that contains some information, such as the client identifier of the component that fired the event (in this case, the zoom-level menu), and the current status of the Ajax request, represented by the poorly named `name` property.

The `zoomChanging()` function shown in Listing 14 calculates the client identifier of the progress bar image and then uses the Prototype `Element` object to hide and show the appropriate HTML elements during the Ajax call.

Conclusion

Over the years, JSF 1 developed a reputation as a framework that was difficult to use. In many respects, that reputation was deserved. JSF 1 was developed in an ivory tower without the considerable hindsight that real-world use affords. As a result, JSF made it much more difficult to implement applications and components than it should have been.

JSF 2, on the other hand, was born from the crucible of real-world experience, by folks who implemented open source projects on top of JSF 1. That real-world hindsight resulted in a much more savvy framework that makes it easy to implement robust, Ajaxified applications.

Throughout this series, I've shown you some of the most prominent JSF 2 features, such as annotations and convention that replace configuration, simplified navigation, support for resources, composite components, built-in Ajax, and the expanded event model. But JSF 2 has many more features that I did not cover in this series, such as View and Page scopes, support for bookmarkable pages, and Project Stage. All those features, and more, make JSF 2 a vast improvement over its predecessor.

Downloads

Description	Name	Size	Download method
Source code for this article's examples	j-jsf2-fu-3.zip	7.7MB	HTTP

Information about download methods

Resources

Learn

- The JSF home page: Find more resources about developing with JSF.
- GMaps4JSF: This library helps JSF users build mashups that use Google Maps.
- Roger Kitain's blog: Roger Kitain and Ed Burns are the co-spec leads for JSF 2.0.
- Jim Driscoll's blog: You'll find numerous entries pertaining to JSF 2.
- Ryan Lubke's blog: Ryan Lubke works on the JSF 2 reference implementation.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- JSF: Download JSF 2.0.
- GMaps4JSF: Download GMaps4JSF

Discuss

- "Public Access to JSF 2.0 JSR-314-EG Discussions Now Available" (Ed Burns's Blog, java.net, March 2009): Find out how to register for the JSF 2 Expert Group mailing list.
- Get involved in the developerWorks community.

About the author

David Geary
Author, speaker, and consultant David Geary is the president of Clarity Training, Inc., where he teaches developers to implement Web applications using JSF and Google Web Toolkit (GWT). He was on the JSTL 1.0 and JSF 1.0/2.0 Expert Groups, co-authored Sun's Web Developer Certification Exam, and has contributed to open source projects, including Apache Struts and Apache Shale. David's *Graphic Java Swing* was one of the best-selling Java books of all time, and *Core JSF* (co-written with Cay Horstman), is the best-selling JSF book. David also speaks regularly at conferences and user groups. He has been a regular on the NFJS tour since 2003, is a three-time Java University instructor, and was twice voted a JavaOne rock star.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.