JSF 2 fu, Part 2: Templating and composite components

Implement extensible UIs with JavaServer Faces 2

Skill Level: Intermediate

David Geary
President
Clarity Training, Inc.

02 Jun 2009

Java™Server Faces (JSF) 2 lets you implement user interfaces that are easy to modify and extend with two powerful features: templating and composite components. In this article — second in a three-part series on JSF 2's new features — JSF 2 Expert Group member David Geary shows you how your Web applications can best take advantage of templating and composite components.

Back in 2000, when I was active on a JavaServer Pages (JSP) mailing list, I came across Craig McClanahan, who was working on a nascent Web framework called Struts. Back then, while moving from Swing to server-side Java programming, I'd implemented a small framework that separated a JSP view's layout from its content, similar in spirit to Swing's layout managers. Craig asked if I'd like to include my *templating* library in Struts, and I readily agreed. The Struts Template Library, bundled with Struts 1.0, became the basis for Struts' popular Tiles library, which eventually became a top-level Apache framework.

Today, JSF 2's default display technology — Facelets — is a templating framework based to a great degree on Tiles. JSF 2 also provides a powerful mechanism called *composite components*, which builds on Facelets' templating features so that you can implement custom components with no Java code and no XML configuration. In this article, I'll introduce you to templating and composite components, with three tips for getting the most out of JSF 2:

- Tip 1: Stay DRY

- Tip 2: Be composed
- Tip 3: Think LEGOs

Facelets and JSF 2

While standardizing the open source Facelets implementation, the JSF 2 Expert Group made some changes to the underlying Facelets API but retained backward-compatibility with the tag library. That means that existing views implemented with the open source version of Facelets should work with JSF 2.

You can find out more about Facelets' many features in Rick Hightower's articles "Facelets Fits JSF like a Glove" and "Advanced Facelets programming."

Tip 1: Stay DRY

In my first job as a software developer, I implemented a GUI for UNIX®-based computer-aided design and computer-aided manufacturing (CAD/CAM) systems.

All went reasonably well initially, but over time my code became more and more problematic. By the time we released, the system was brittle enough that I dreaded fixing bugs, and the release precipitated a steady flow of bug reports.

If I had followed the DRY principle — Don't Repeat Yourself — on that project, I could have saved myself a lot grief. The DRY principle, coined by Dave Thomas and Andy Hunt (see Resources), states:

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

My CAD/CAM application wasn't DRY — it had too much coupling between concerns — so changes in one area caused unexpected changes elsewhere.

JSF 1 violated the DRY principle in several respects, for example by forcing you to provide two representations of your managed beans — one in XML, and another in Java code. The need for multiple representations made it more difficult to create and change managed beans. As I showed you in Part 1, JSF 2 lets you use annotations instead of XML to configure managed beans, giving you a single, authoritative representation of your managed beans.

Managed beans aside, even practices that might seem benign — such as including the same stylesheet in all of your views — violate the DRY principle and can cause consternation. If you change that stylesheet's name, you must change multiple views. It's better to encapsulate the style-sheet inclusion if you can.

The DRY principle also applies to your code's design. If you have multiple methods that all contain code to walk a tree, for example, it's a good idea to encapsulate the tree-walking algorithm, perhaps in a subclass.

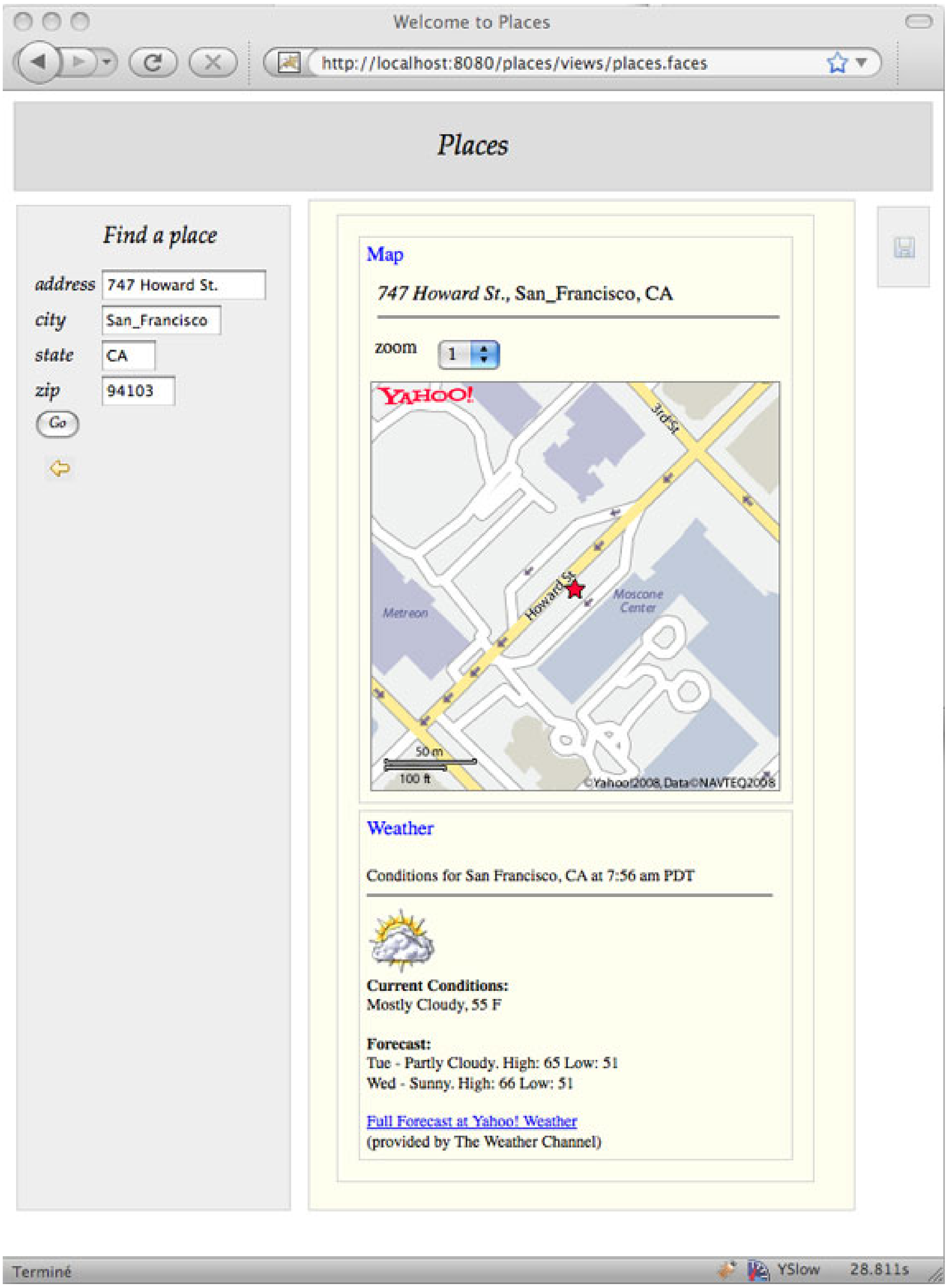
It's especially important to stay DRY when you implement a UI, where (arguably) the most changes occur during development.

JSF 2 templating

One of the many ways in which JSF 2 supports the DRY principle is *templating*. Templates encapsulate functionality that's common among views in your application, so you need to specify that functionality only once. One template is used by multiple *compositions* to create views in a JSF 2 application.

The places application, which I introduced in Part 1, has the three views shown in Figure 1:

Figure 1. The places application's views: Login, source viewer, and places



Welcome to Places

http://localhost:8080/places/views/places.faces

Places

Find a place

address	747 Howard St.
city	San_Francisco
state	CA
zip	94103

Go

Map

747 Howard St., San_Francisco, CA

zoom 1

Weather

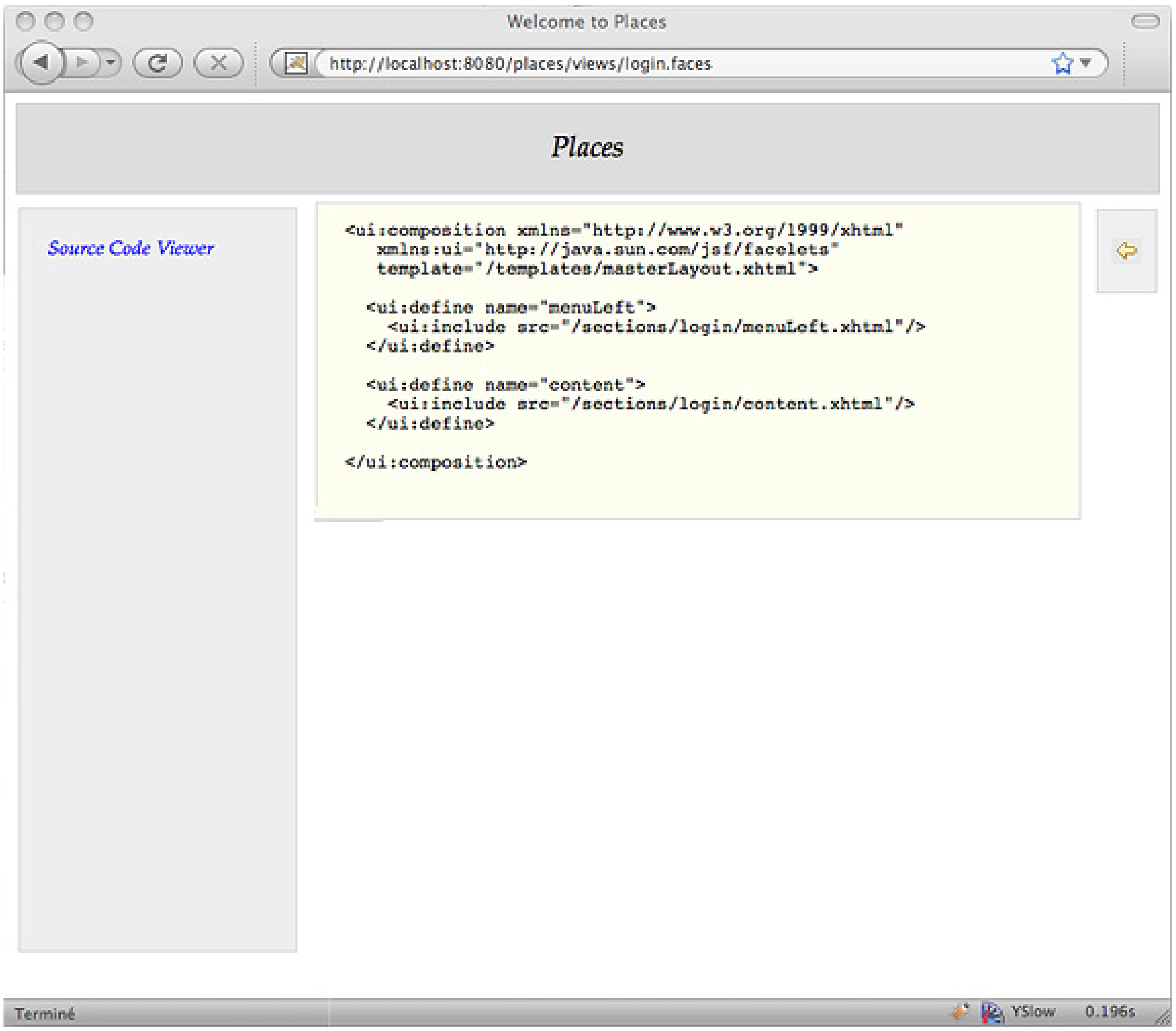
Conditions for San Francisco, CA at 7:56 am PDT

Current Conditions:
Mostly Cloudy, 55 F

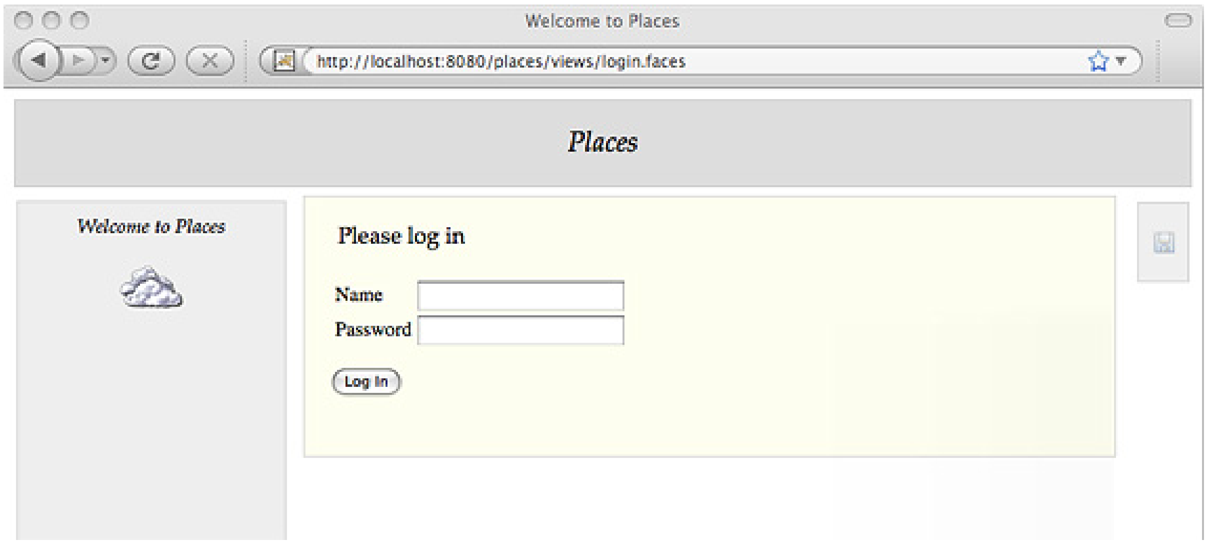
Forecast:
Tue - Partly Cloudy. High: 65 Low: 51
Wed - Sunny. High: 66 Low: 51

Full Forecast at Yahoo! Weather
(provided by The Weather Channel)

Terminé YSlow 28.811s



```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   template="/templates/masterLayout.xhtml">

  <ui:define name="menuLeft">
    <ui:include src="/sections/login/menuLeft.xhtml"/>
  </ui:define>

  <ui:define name="content">
    <ui:include src="/sections/login/content.xhtml"/>
  </ui:define>

</ui:composition>
```



Like many Web applications, the places application contains multiple views that share the same layout. JSF templating lets you encapsulate that layout — along with other shared artifacts, such as JavaScript and Cascading Style Sheets (CSS) — in a template. Listing 1 is the template for the three views shown in Figure 1:

Listing 1. The places template: /templates/masterLayout.xhtml

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

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

  <h:head>
    <title>
      <ui:insert name="windowTitle">
        #{msgs.placesWindowTitle}
      </ui:insert>
    </title>
  </h:head>

  <h:body>
    <h:outputScript library="javascript" name="util.js" target="head"/>
    <h:outputStylesheet library="css" name="styles.css" target="body"/>

    <div class="pageHeading">
      <ui:insert name="heading">
        #{msgs.placesHeading}
      </ui:insert>
    </div>

    <div class="menuAndContent">
      <div class="menuLeft">
        <ui:insert name="menuLeft"/>
      </div>

      <div class="content" style="display: #{places.showContent}">
        <ui:insert name="content"/>
      </div>

      <div class="menuRight">
        <ui:insert name="menuRight">
          <ui:include src="/sections/shared/sourceViewer.xhtml"/>
        </ui:insert>
      </div>
    </div>
  </h:body>
</html>
```

The template in Listing 1 provides the following infrastructure for all the application's views:

- HTML `<head>`, `<body>`, and `<title>`
- A default title (which can be overridden by compositions that use the template)
- A CSS stylesheet

- Some utility JavaScript
- A layout in the form of `<div>`s and their corresponding CSS classes
- Default content for the heading (which can be overridden)
- Default content for the right menu (which can be overridden)

As Listing 1 illustrates, templates insert content into their layout with the `<ui:insert>` tag.

If you specify a body for a `<ui:insert>` tag, as I've done for the window title, heading, and right menu in Listing 1, JSF uses the body of the tag as the *default content*. Compositions that use the template can define content, or override default content, with the `<ui:define>` tag, as evidenced by Listing 2, which shows the markup for the login view:

Listing 2. The login view

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   template="/templates/masterLayout.xhtml">

  <ui:define name="menuLeft">
    <ui:include src="/sections/login/menuLeft.xhtml"/>
  </ui:define>

  <ui:define name="content">
    <ui:include src="/sections/login/content.xhtml"/>
  </ui:define>

</ui:composition>
```

The login view uses the template's default content for the window title, heading, and right menu. It defines only the functionality that's specific to the login view: the content section and the left menu.

By providing a `<ui:define>` tag for the window title, heading, or right menu, I could have overridden the default content that the template defines. For example, Listing 3 shows the source-viewer view (the middle picture in Figure 1):

Listing 3. The source-viewer view

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   template="/templates/masterLayout.xhtml">

  <ui:define name="content">
    <ui:include src="/sections/showSource/content.xhtml"/>
  </ui:define>

  <ui:define name="menuLeft">
    <ui:include src="/sections/showSource/menuLeft.xhtml"/>
  </ui:define>
  <ui:define name="menuRight">
    <ui:include src="/sections/showSource/menuRight.xhtml"/>
  </ui:define>

</ui:composition>
```

The source-viewer view defines content for the content section and for the right menu. It also overrides the default content, defined by the template in Listing 1, for the left menu.

Listing 4 shows the places view (the bottom picture in Figure 1):

Listing 4. The places view

```

<ui:composition xmlns="http://www.w3.org/1999/xhtml"
   xmlns:ui="http://java.sun.com/jsf/facelets"
   template="/templates/masterLayout.xhtml">

   <ui:define name="menuLeft">
    <ui:include src="/sections/places/menuLeft.xhtml"/>
   </ui:define>

  <ui:define name="content">
    <ui:include src="/sections/places/content.xhtml"/>
  </ui:define>

</ui:composition>
```

JSF 2 templating

The concept behind templating is simple. You define a single template that encapsulates common functionality among multiple views. Each view consists of a composition and a template.

When JSF creates a view, it loads the composition's template and then inserts the content defined by the composition into the template.

Notice the similarities among Listings 2, 3, and 4. All three views specify their template and define content. Also notice how easy it is to create new views, because most of the view infrastructure is encapsulated in a template and included files.

Another interesting aspect of using JSF templating is that views like those in Listings 2, 3, and 4 do not change much over time, so a good portion of your view code requires essentially no maintenance.

Like the views that use them, templates also tend to change very little. And because you've encapsulated so much common functionality in nearly maintenance-free code, you can concentrate on the actual content of your views — what goes in the left menu of the login page, for example. Concentrating on your views' actual content is what the next tip is all about.

Tip 2: Be composed

Not long after my CAD/CAM GUI was released, I spent a few months working on an unrelated project with a developer named Bob. We were working on his code base, and amazingly, we were able to make changes and fix bugs with ease.

I soon realized that the single biggest difference between Bob's code and mine was that he wrote *tiny* methods — typically between 5 and 15 lines of code — and his entire system was cobbled together from those methods. Whereas I had struggled to modify long methods with many concerns on my previous project, Bob nimbly combined small methods with atomic functionality. The difference in maintainability and extensibility between Bob's code and mine was like night and day, and from then on I was sold on tiny methods.

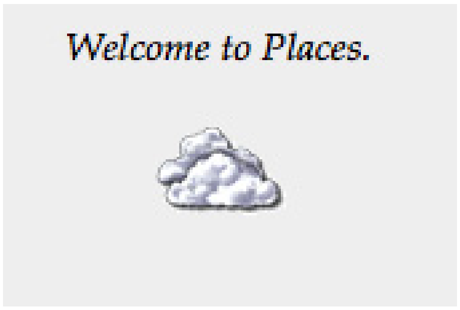
Neither Bob nor I knew it back then, but we were using a design pattern from Smalltalk called Composed Method (see Resources):

> *Divide your software into methods that perform one identifiable task, at a single level of abstraction.*

The benefits of using the Composed Method pattern are well-documented. (See Neal Ford's "Evolutionary architecture and emergent design: Composed method and SLAP" for an excellent detailed explanation.) Here, I'll focus on how you can use the Composed Method pattern with your JSF views.

JSF 2 encourages you to compose your views out of smaller view fragments. Templating encapsulates common functionality, thus dividing your views into smaller pieces. JSF 2 also provides a `<ui:include>` tag, as I demonstrated in the preceding listings, that lets you further divide your views into smaller pieces of functionality. For example, Figure 2 shows the left menu of the places application's login page:

Figure 2. The login page's left menu



And Listing 5 shows the file that defines that menu's content:

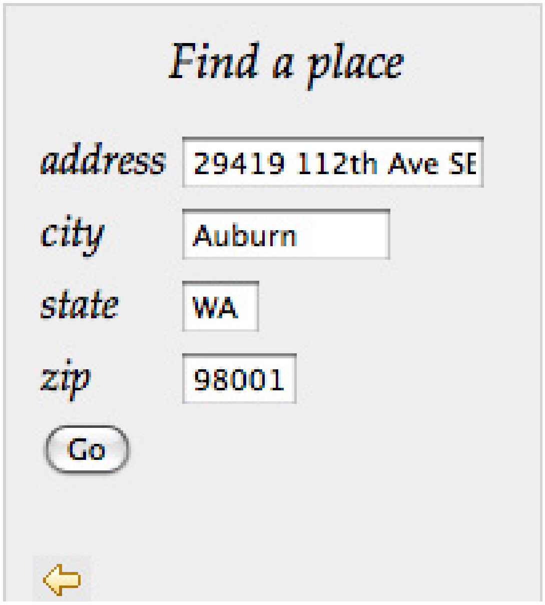
Listing 5. The implementation of the login view's left menu

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

  <div class="menuLeftText">
    #{msgs.welcomeGreeting}

    <div class="welcomeImage">
      <h:graphicImage library="images" name="cloudy.gif"/>
    </div>
  </div>

</html>
```

The markup in Listing 5 is simple, which makes the file easy to read, understand, maintain, and extend. If the same code were buried in a single long XHTML page that contained everything needed to implement the login view, it would be cumbersome to change.

Figure 3 shows the places view's left menu:

Figure 3. The places view's left menu



address: 29419 112th Ave SE
city: Auburn
state: WA
zip: 98001

An implementation of the places view's left menu is shown in Listing 6:

Listing 6. The implementation of the places view's left menu

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    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">

  <div class="placesSearchForm">
    <div class="placesSearchFormHeading">
      #{msgs.findAPlace}
    </div>
    <h:form prependId="false">
      <h:panelGrid columns="2">

        #{msgs.streetAddress}
        <h:inputText value="#{place.streetAddress}" size="15"/>

        #{msgs.city}  <h:inputText value="#{place.city}"  size="10"/>
        #{msgs.state} <h:inputText value="#{place.state}" size="3"/>
        #{msgs.zip}   <h:inputText value="#{place.zip}"   size="5"/>

        <h:commandButton value="#{msgs.goButtonText}"
          style="font-family:Palatino;font-style:italic"
          action="#{place.fetch}"/>

      </h:panelGrid>
    </h:form>
  </div>

  <util:icon image="#{resource['images:back.jpg']}"
    actionMethod="#{places.logout}"
    style="border: thin solid lightBlue"/>

</ui:composition>
```

Listing 6 implements a form, and it uses an icon component. (I'll discuss that icon component in The icon component, shortly. For now, it's enough to know that a page author can associate an image and a method with an icon.) The image for the logout icon is shown at the bottom of Figure 3, and the logout icon's method — `places.logout()` — is shown in Listing 7:

Listing 7. The Places.logout() method

```
package com.clarity;
...
@ManagedBean()
@SessionScoped

public class Places {
  private ArrayList<Place> places = null;
  ...
  private static SelectItem[] zoomLevelItems = {
  ...
  public String logout() {
    FacesContext fc = FacesContext.getCurrentInstance();
    ELResolver elResolver = fc.getApplication().getELResolver();

    User user = (User)elResolver.getValue(
      fc.getELContext(), null, "user");

    user.setName("");
    user.setPassword("");

    setPlacesList(null);

    return "login";
  }
}
```

For me, Listing 6 — the implementation of the places view's left menu — is approaching a too-much-code threshold at around 30 lines of markup. That listing is somewhat difficult to read, and two things in that code could be refactored into their own files: the form and the icon. Listing 8 shows the refactored version of Listing 6 that encapsulates the form and icon into their own XHTML files:

Listing 8. Refactoring the places view's left menu

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

  <div class="placesSearchForm">
    <div class="placesSearchFormHeading">
      #{msgs.findAPlace}
    </div>

    <ui:include src="addressForm.xhtml">
    <ui:include src="logoutIcon.xhtml">
  </div>

</ui:composition>
```

Listing 9 shows addressForm.xhtml:

Listing 9. addressForm.xhtml

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

  <h:form prependId="false">
    <h:panelGrid columns="2">

      #{msgs.streetAddress}
      <h:inputText value="#{place.streetAddress}" size="15"/>

      #{msgs.city}  <h:inputText value="#{place.city}"  size="10"/>
      #{msgs.state} <h:inputText value="#{place.state}" size="3"/>
      #{msgs.zip}   <h:inputText value="#{place.zip}"   size="5"/>

      <h:commandButton value="#{msgs.goButtonText}"
        style="font-family:Palatino;font-style:italic"
        action="#{place.fetch}"/>

    </h:panelGrid>
  </h:form>

</ui:composition>
```

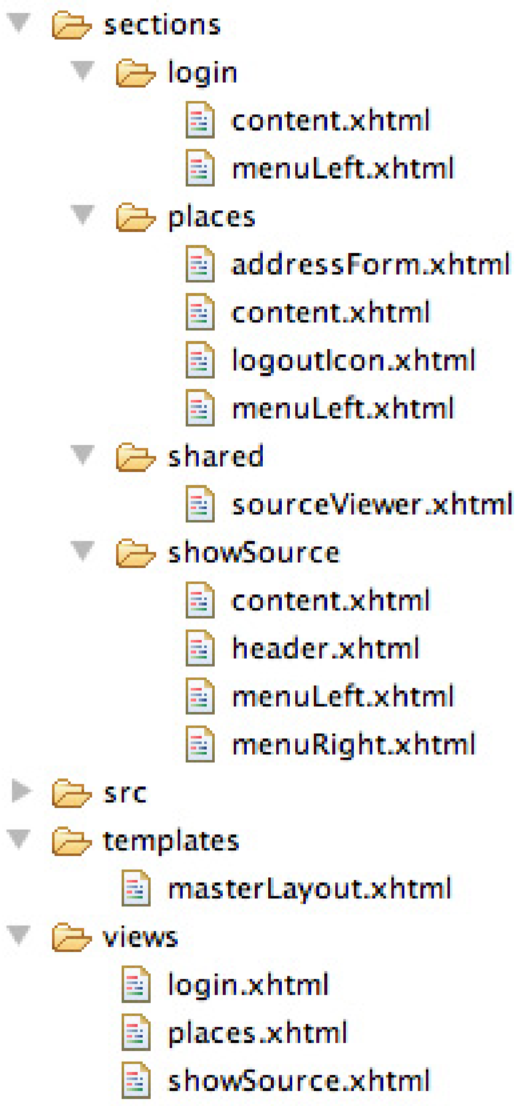
Listing 10 shows logoutIcon.xhtml:

Listing 10. logoutIcon.xhtml

```
<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:icon image="#{resource['images:back.jpg']}"
    actionMethod="#{places.logout}"
    style="border: thin solid lightBlue"/>

</ui:composition>
```

When you compose your views from multiple small files, you reap the benefits of Smalltalk's Composed Method pattern. You can also organize your files to make it even easier to react to change. For example, Figure 4 shows the files that constitute the three views in the places application:

Figure 4. The places application's views



```
sections
    login
        content.xhtml
        menuLeft.xhtml
    places
        addressForm.xhtml
        content.xhtml
        logoutIcon.xhtml
        menuLeft.xhtml
    shared
        sourceViewer.xhtml
    showSource
        content.xhtml
        header.xhtml
        menuLeft.xhtml
        menuRight.xhtml
src
templates
    masterLayout.xhtml
views
    login.xhtml
    places.xhtml
    showSource.xhtml
```

Three directories I've created — views, sections, and templates — contain most of the XHTML files used to implement the places application's views. Because the files in the views and templates directories rarely change, I can concentrate on the sections directory. If I want to change the icon in the left menu of the login page, for example, I know exactly where to go: sections/login/menuLeft.xhtml.

You can use whatever directory structure you want to organize your XHTML files. A logical structure makes it easy to locate the code that you need to modify.

Besides adhering to the DRY principle and using the Composed Method pattern, it's also a good idea to encapsulate functionality in custom components. Components are a powerful reuse mechanism, and you should take advantage of that power. Unlike JSF 1, JSF 2 makes it easy to implement custom components.

Tip 3: Think LEGOs

As a boy, I had two favorite toys: a chemistry set and LEGOs. Both let me create things by combining fundamental building blocks, something that has become a lifelong fascination in the guise of software development.

JSF's strength has always been its component model, but that strength wasn't fully realized until now because it was difficult to implement custom components with JSF 1. You had to write Java code, specify XML configuration, and have a good grasp of JSF's life cycle. With JSF 2, you can implement custom components:

- With no configuration, XML or otherwise.
- With no Java code.
- To which developers can attach functionality.
- That hot-deploy when modified.

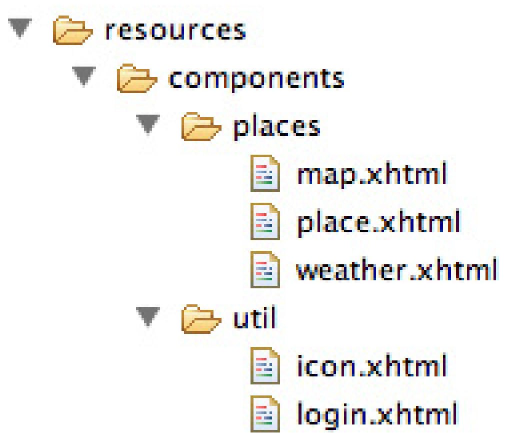
For the rest of this article, I'll walk you through the implementation of three custom components for the places application: an icon, a login panel, and a panel that shows an address's map and weather information. But first, I'll give you an overview of JSF 2 composite components.

Implementing custom components

JSF 2 combines Facelets templating, resource handling (discussed in Part 1), and a simple naming convention to implement *composite components*. Composite components, as the name indicates, let you compose a component from existing components.

You implement composite components in XHTML somewhere under the resources directory and link them, purely by convention, to a namespace and a tag. Figure 5 shows how I've organized the composite components for the places application:

Figure 5. The places application's components



To use composite components, you declare a namespace and use the tags. The namespace is always `http://java.sun.com/jsf/composite` plus the name of the directory in which the component resides under the resources directory. The name of the component itself is the name of its XHTML file, without the .xhtml extension. This convention obviates the need for any configuration whatsoever. For example, to use the `login` component in the places application, you would do this:

```
<html xmlns="http://www.w3.org/1999/xhtml"
    ...
    xmlns:util="http://java.sun.com/jsf/composite/component/util">
  ...
  <util:login.../>
  ...
<html>
```

And to use the `icon` component, you would do this:

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

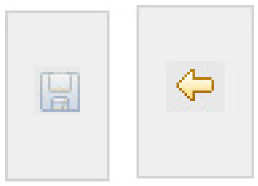
Finally, you use the place component like this:

```
<html xmlns="http://www.w3.org/1999/xhtml"
    ...
    xmlns:util="http://java.sun.com/jsf/composite/components/places">
  ...
  <places:place.../>
  ...
<html>
```

The icon component: A simple composite component

The places application uses the two icons shown in Figure 6:

Figure 6. The places application's icons



Each icon is a link. When a user clicks the left icon in Figure 6, JSF shows the markup for the current view, whereas activating the right icon logs the user out of the application.

You can specify a CSS classname and an image for links, and you can also attach methods to links. JSF invokes those methods when the user clicks on an associated link.

Listing 11 shows how the `icon` component is used in the places application to show markup:

Listing 11. Using the icon component to show markup

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

  <util:icon actionMethod="#{sourceViewer.showSource}"
                      image="#{resource['images:disk-icon.jpg']}"/>
  ...
</html>
```

Listing 12 shows how the `icon` component is used to logout:

Listing 12. Using the icon component to log out

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

  <util:icon actionMethod="#{places.logout}"
                      image="#{resource['images:back-arrow.jpg']}"/>
  ...
</html>
```

Listing 13 shows the code for the `icon` component:

Listing 13. The icon component

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

  <!-- INTERFACE -->
  <composite:interface>
    <composite:attribute name="image"/>
    <composite:attribute name="actionMethod"
              method-signature="java.lang.String action()"/>
  </composite:interface>

  <!-- IMPLEMENTATION -->
    <composite:implementation>
    <h:form>
      <h:commandLink action="#{cc.attrs.actionMethod}" immediate="true">

      <h:graphicImage value="#{cc.attrs.image}"
                styleClass="icon"/>

      </h:commandLink>
    </h:form>
  </composite:implementation>
</html>
```

Like all composite components, the `icon` component in Listing 13 contains two sections: `<composite:interface>` and `<composite:implementation>`. The `<composite:interface>` section defines an interface that you can use to configure the component. The `icon` component has two attributes: `image`, which defines what the component looks like, and `actionMethod`, which defines how it behaves.

The `<composite:implementation>` section contains the component's implementation. It uses the `#{cc.attrs.`*`ATTRIBUTE_NAME`*`}` expression to access attributes defined in the component's interface. (The `cc`, which is a reserved keyword in the JSF 2 expression language, stands for composite component.)

Notice that the `icon` component in Listing 13 specifies a CSS class for its image with `<h:graphicImage>`'s `styleClass` attribute. The name of that CSS class is hardcoded as `icon`, so you can just specify a CSS class with that name and JSF will use that class for all of the icons in an application. But what if you want to override that CSS classname? In that case, I could add another attribute for the CSS classname and provide a default that will be used by JSF when the attribute is not specified. Listing 14 shows what that attribute would look like:

Listing 14. The icon component, refactored

```
<!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">

    <composite:interface>
      ...
      <composite:attribute name="styleClass" default="icon" required="false"/>
      ...
    </composite:interface>

    <composite:implementation>
      ...
      <h:graphicImage value="#{cc.attrs.image}"
                styleClass="#{cc.attrs.styleClass}"/>
      ...
    </composite:implementation>
</html>
```

In Listing 14, I've added an attribute to the icon component's interface named `styleClass` and referenced that attribute in the component's implementation. With that change, you can now specify an optional CSS class for the icon's image, like this:

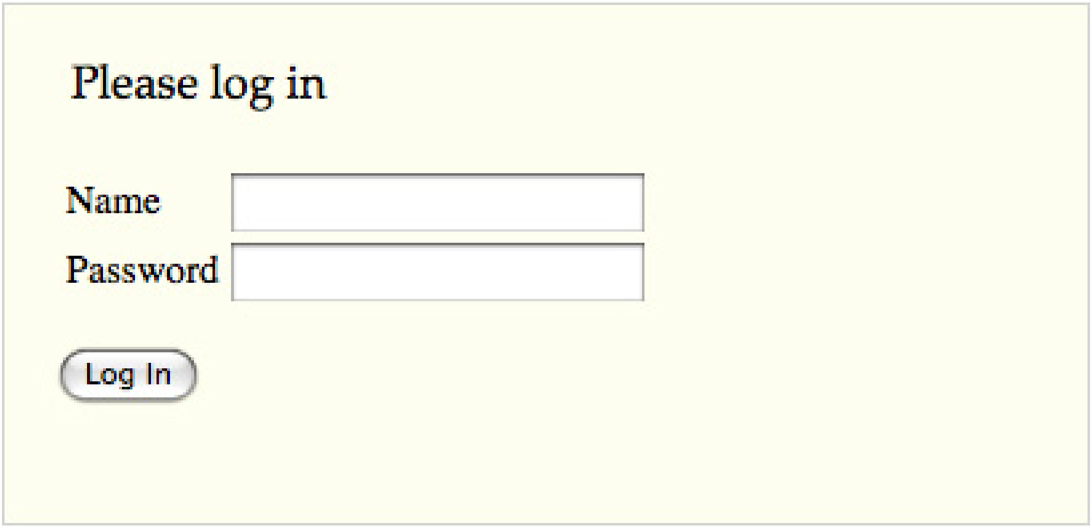
```
<util:icon actionMethod="#{places.logout}"
                  image="#{resource['images:back-arrow.jpg']}"
            styleClass="customIconClass"/>
```

If you don't specify the `styleClass` attribute, JSF will use the default value, `icon`.

The login component: A fully configurable component

With JSF 2, you can implement fully configurable composite components. For example, the places application contains a `login` component, shown in Figure 7:

Figure 7. The places application's login component



Listing 15 shows how the places application uses the `login` component:

Listing 15. Using the login component

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

<html xmlns="http://www.w3.org/1999/xhtml"
  ...
  xmlns:comp="http://java.sun.com/jsf/composite/component/util">

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

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

  </util:login>
  ...
</html>
```

Not only does Listing 15 parameterize the `login` component's attributes, such as the name and password prompts, but it also attaches an action listener to the component's **Log In** button. That button is exposed by the `login` component's interface, as shown in Listing 16:

Listing 16. The login component

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

<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>
    <composite:actionSource name="loginButton" targets="form:loginButton"/>
    <composite:attribute name="loginButtonText" default="Log In" required="true"/>
    <composite:attribute name="loginPrompt"/>
    <composite:attribute name="namePrompt"/>
    <composite:attribute name="passwordPrompt"/>
    <composite:attribute name="loginAction"
      method-signature="java.lang.String action()"/>
    <composite:attribute name="managedBean"/>
  </composite:interface>

  <!-- IMPLEMENTATION -->
  <composite:implementation>
   <h:form id="form" prependId="false">

     <div class="prompt">
       #{cc.attrs.loginPrompt}
     </div>

     <panelGrid columns="2">
       #{cc.attrs.namePrompt}
       <h:inputText id="name" value="#{cc.attrs.managedBean.name}"/>

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

     </panelGrid>

     <p>
       <h:commandButton id="loginButton"
                        value="#{cc.attrs.loginButtonText}"
                       action="#{cc.attrs.loginAction}"/>
     </p>
   </h:form>

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

In the interface of the login component, I've exposed the **Log In** button under the name `loginButton`. That name targets the **Log In** button that resides in the form named `form`, thus the value of the `targets` attribute: `form:loginButton`.

The action listener associated with the **Log In** button in Listing 16 is shown in Listing 17:

Listing 17. The Log In button's action listener

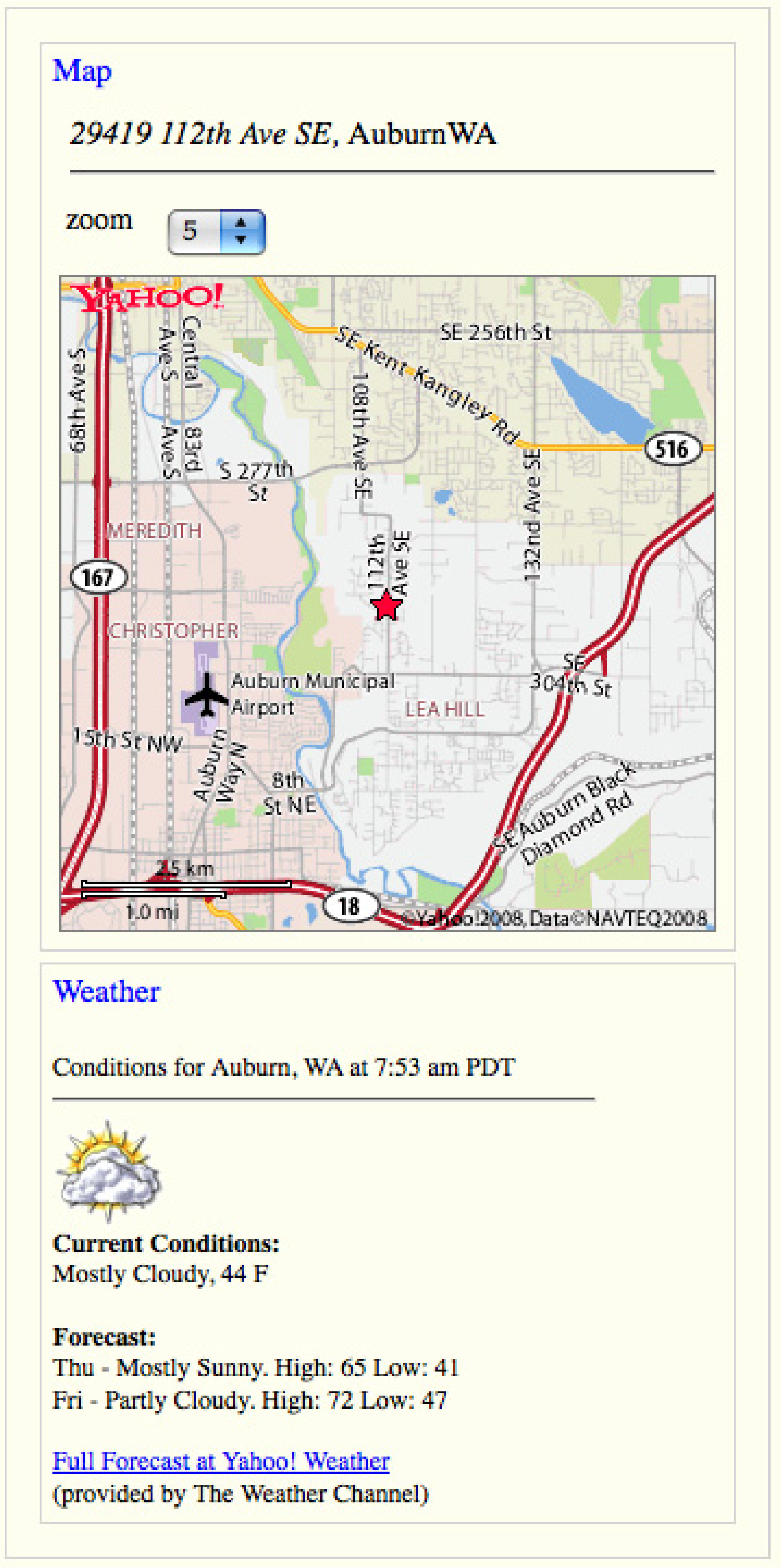
```
package com.clarity;

import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import javax.faces.event.ActionListener;

public class LoginActionListener implements ActionListener {
  public void processAction(ActionEvent e)
    throws AbortProcessingException {
    System.out.println("logging in ..........");
  }
}
```

The action listener in Listing 17 is for illustrative purposes only — when the user logs in, I simply write out a message to the servlet container log file. But you get the idea: With JSF 2, you can implement fully configurable components, and you can attach functionality to those components, all without a single line of Java code or XML configuration. That's some powerful fu.

The place component: Nesting composite components

JSF 2 lets you implement fully configurable components without any Java code or configuration. You can also nest composite components, which lets you break complex components into smaller, more manageable pieces. For example, Figure 8 shows the `place` component, which displays a map and weather information for a given address:

Figure 8. The places application's place component



Map

29419 112th Ave SE, AuburnWA

zoom 5

Weather

Conditions for Auburn, WA at 7:53 am PDT

Current Conditions:
Mostly Cloudy, 44 F

Forecast:
Thu - Mostly Sunny. High: 65 Low: 41
Fri - Partly Cloudy. High: 72 Low: 47

Full Forecast at Yahoo! Weather
(provided by The Weather Channel)

Listing 18 shows how the places application uses the `place` component:

Listing 18. Using the place component

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

  <h:form id="form">
    <ui:repeat value="#{places.placesList}" var="place">
      <places:place location="#{place}"/>
    </ui:repeat>
  </h:form>
</ui:composition>
```

The code for the `place` component is shown in Listing 19:

Listing 19. The place component

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

<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="location" required="true"/>
  </composite:interface>

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

      <places:map     title="Map"/>
     <places:weather title="Weather"/>

    </div>
  </composite:implementation>

</html>
```

In Listing 19, the `place` component uses two nested components: `<places:map>` and `<places:weather>`. Listing 20 shows the `map` component:

Listing 20. The map component

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

<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:ui="http://java.sun.com/jsf/facelets"
    xmlns:composite="http://java.sun.com/jsf/composite">

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

   <!-- IMPLEMENTATION -->
   <composite:implementation>
      <div class="map">
      <div style="padding-bottom: 10px;">
        <h:outputText value="#{cc.attrs.title}"
                      style="color: blue"/>
      </div>

      <h:panelGrid columns="1">
       <h:panelGroup>
         <div style="padding-left: 5px;">
           <i>
             <h:outputText value="#{cc.parent.attrs.location.streetAddress}, "/>
           </i>

           <h:outputText value=" #{cc.parent.attrs.location.city}" />
           <h:outputText value="#{cc.parent.attrs.location.state}"/><hr/>
         </div>
       </h:panelGroup>

       <h:panelGrid columns="2">
         <div style="padding-right: 10px;margin-bottom: 10px;font-size:14px">
           #{msgs.zoomPrompt}
         </div>

         <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="#{cc.parent.attrs.location.zoomLevelItems}"/>

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

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

     </h:panelGrid>
     </div>
   </composite:implementation>
</html>
```

Composite-component refactoring

Listing 20 — the markup for the `map` component — is a bit long for my taste. It's somewhat difficult to understand at first glance, and its complexity could pose problems later on.

You can easily refactor Listing 20 into multiple, more manageable files, as I did earlier when I refactored the places view's left menu in Listings 8, 9, and 10. In this case, I'll leave the refactoring as an exercise for you.

Notice the use of the expression `#{cc.parent.attrs.location.`*`ATTRIBUTE_NAME`*`}` in Listing 20. You can use a composite component's `parent` attribute to access attributes of the parent component, which greatly facilitates nesting of components.

But you don't need to rely strictly on parent attributes in nested components. As I've done in the place component in Listing 19, you can pass attributes, such as the map's title, from a parent to its nested component, just as you would pass attributes to any other component, nested or not.

It's somewhat anticlimactic, but Listing 21 shows the `weather` component:

Listing 21. The weather component

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

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

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

  <!-- IMPLEMENTATION -->
  <composite:implementation>

   <div class="weather">
     <div style="padding-bottom: 10px;">
       <h:outputText value="#{cc.attrs.title}"
         style="color: blue"/>
     </div>

     <div style="margin-top: 10px;width:250px;">
       <h:outputText style="font-size: 12px;"
                     value="#{cc.parent.attrs.location.weather}"
                    escape="false"/>
     </div>
   </div>

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

The `weather` component, like the `map` component, uses both a parent component attribute (the weather HTML from the weather Web service), and a component-specific attribute (the title). (See Part 1 to see how the application obtains map and weather information for a particular location.)

So, when you implement nested components, you have a choice. You can let the nested component rely on attributes of its parent component, or you can require the parent component to pass attributes explicitly to the nested component. For example, the `place` component in Listing 19 explicitly passes the title attributes to its nested components, but the nested components rely on the parent's attributes, such as the map URL and the weather HTML.

Whether you choose to implement component-explicit attributes or rely on parent attributes is a trade-off between coupling and convenience. In this case, the `map` and `weather` components are tightly coupled to their parent component (the `place` component) because they rely on the parent component's attributes. I could've decoupled the `map` and `weather` components from the `place` component by specifying all of the `map` and `weather` components' attributes as component-explicit attributes. But in that case, I lose some convenience, because the `place` component would need to pass all of the attributes explicitly to the `map` and `weather` components.

Next time...

In this article, I've shown you how to use JSF 2 to implement UIs that are easy to maintain and extend through templating and composite components. The final article in this series will show you how to use JavaScript in composite components, how to use JSF 2's new event model, and how to take advantage of JSF 2's built-in support for Ajax.

Downloads

Description	Name	Size	Download method
Source code for this article's examples	jsf2fu2.zip	7.4MB	HTTP

Information about download methods

Resources

Learn

- The JSF homepage: Find more resources about developing with JSF.
- "Facelets fits JSF like a glove" (Richard Hightower, developerWorks, February 2006) and "Advanced Facelets programming" (Richard Hightower, developerWorks, May 2006): Learn more about Facelets' features.
- *The Pragmatic Programmer* (Andy Hunt and Dave Thomas, The Pragmatic Bookshelf, 2001): Hunt and Thomas introduced the DRY principle in this book.
- *Smalltalk Best Practice Patterns* (Kent Beck, Prentice Hall, 1996): Learn more about the Composed Method pattern.
- 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.

Discuss

- "Public Access to JSF 2.0 JSR-314-EG Discussions Now Available" (Ed Burns's Blog, java.net, March 2009): Register for the JSF 2 Expert Group mailing list.
- Check out developerWorks blogs and get involved in the My 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