JSF 2 fu, Part 1: Streamline Web application development

Simplify navigation, eliminate XML configuration, and easily access resources with JSF 2

Skill Level: Intermediate

David Geary
President
Clarity Training, Inc.

12 May 2009

With version 2.0, Java™Server Faces (JSF) makes it easy to implement robust, Ajaxified Web applications. This article launches a three-part series by JSF 2.0 Expert Group member David Geary showing you how to take advantage of the new features in JSF 2 to sharpen your skills like a kung fu master. In this installment, you'll learn how to streamline development with JSF 2 by replacing XML configuration with annotations and convention, simplifying navigation, and easily accessing resources. And you'll see how to use Groovy in your JSF applications.

There's an ongoing debate over the best birthplace for Web application frameworks: ivory towers — where eggheads put their heads together — versus the real world, where frameworks are born out of crucibles of burning need. It seems rather intuitive that crucibles of burning need win out over eggheads, and I suppose that intuition stands up well to closer inspection.

JSF 1 was developed in an ivory tower, and the results, arguably, were less than spectacular. But JSF got one thing right — it let the marketplace come up with lots of real-world innovations. Early on, Facelets made its debut as a powerful replacement for JavaServer Pages (JSP). Then came Rich Faces, a slick JSF Ajax library; ICEFaces, a novel approach to Ajax with JSF; Seam; Spring Faces; Woodstock components; JSF Templating; and so on. All of those open source JSF projects were built by developers who needed the functionality they implemented.

The JSF 2.0 Expert Group essentially standardized some of the best features from those open source projects. Although JSF 2 was indeed specified by a bunch of eggheads, it was also driven by real-world innovation. In retrospect, the expert group's job was relatively easy because we were standing on the shoulders of giants such as Gavin King (Seam), Alexandr Smirnov (Rich Faces), Ted Goddard (ICEFaces), and Ken Paulson (JSF Templating). In fact, all of those giants were on the JSF 2 Expert Group. So in many respects, JSF 2 combines the best aspects of ivory tower and real world. And it shows. JSF 2 is a vast improvement over its progenitor.

This is the first article in a three-part series that has two goals: to show you exciting JSF 2 features, and to show you how to best use those features, so that you can take advantage of what JSF 2 offers. I will crosscut those two concerns by illustrating the use of JSF 2 with some tips for best using JSF. Here are the tips for this article:

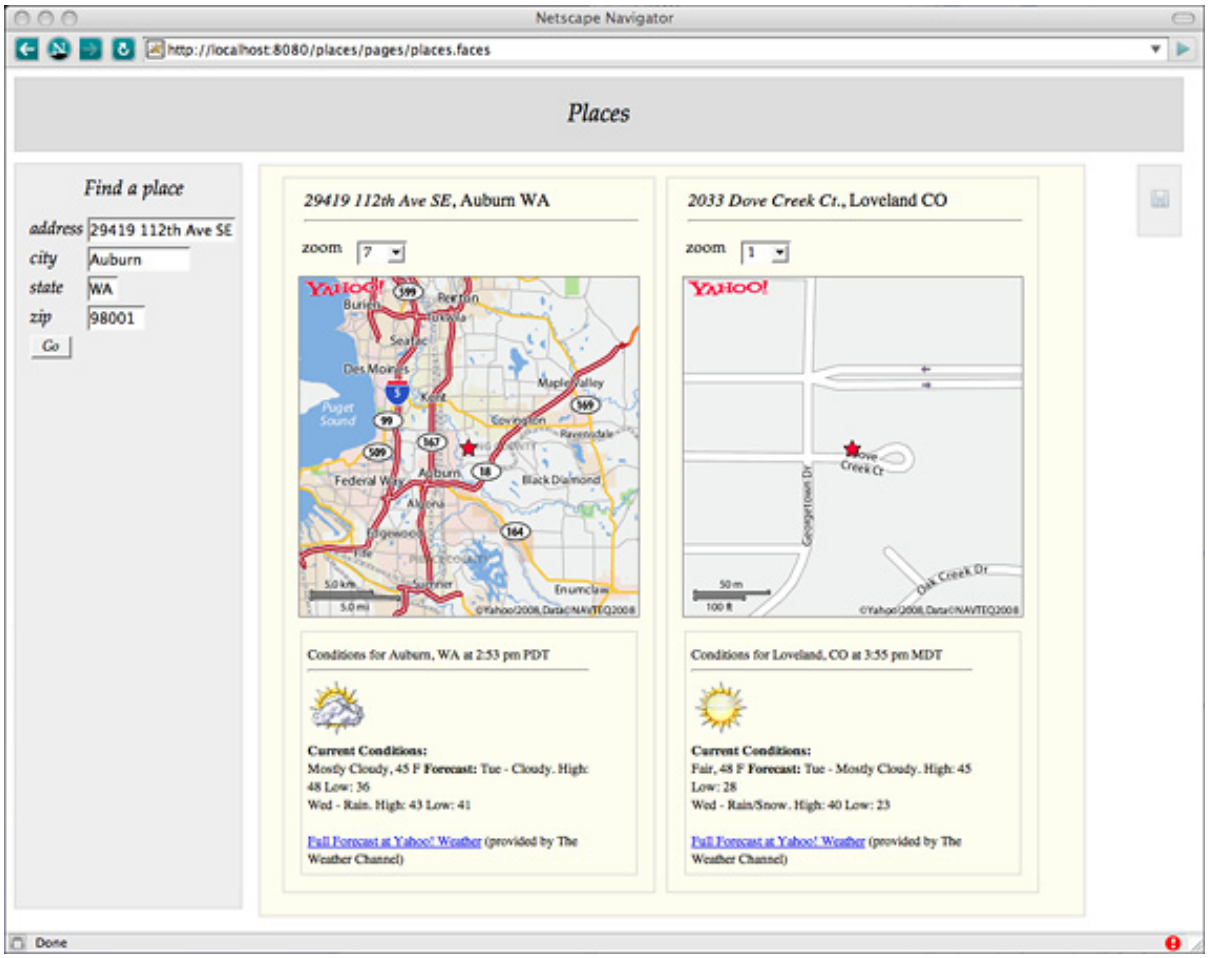
- Tip 1: Get rid of XML configuration
- Tip 2: Simplify navigation
- Tip 3: Use Groovy
- Tip 4: Take advantage of resource handling

First, though, I'll introduce the example application that I use throughout this article series. The application source code for this article is available for download.

The obligatory mapping-based Web services mashup example

Figure 1 shows a JSF mashup — I'll refer to it as the places application — that uses Yahoo! Web services to convert addresses into maps with zoom levels and weather forecasts:

Figure 1. Viewing maps and weather information from Yahoo! Web Services

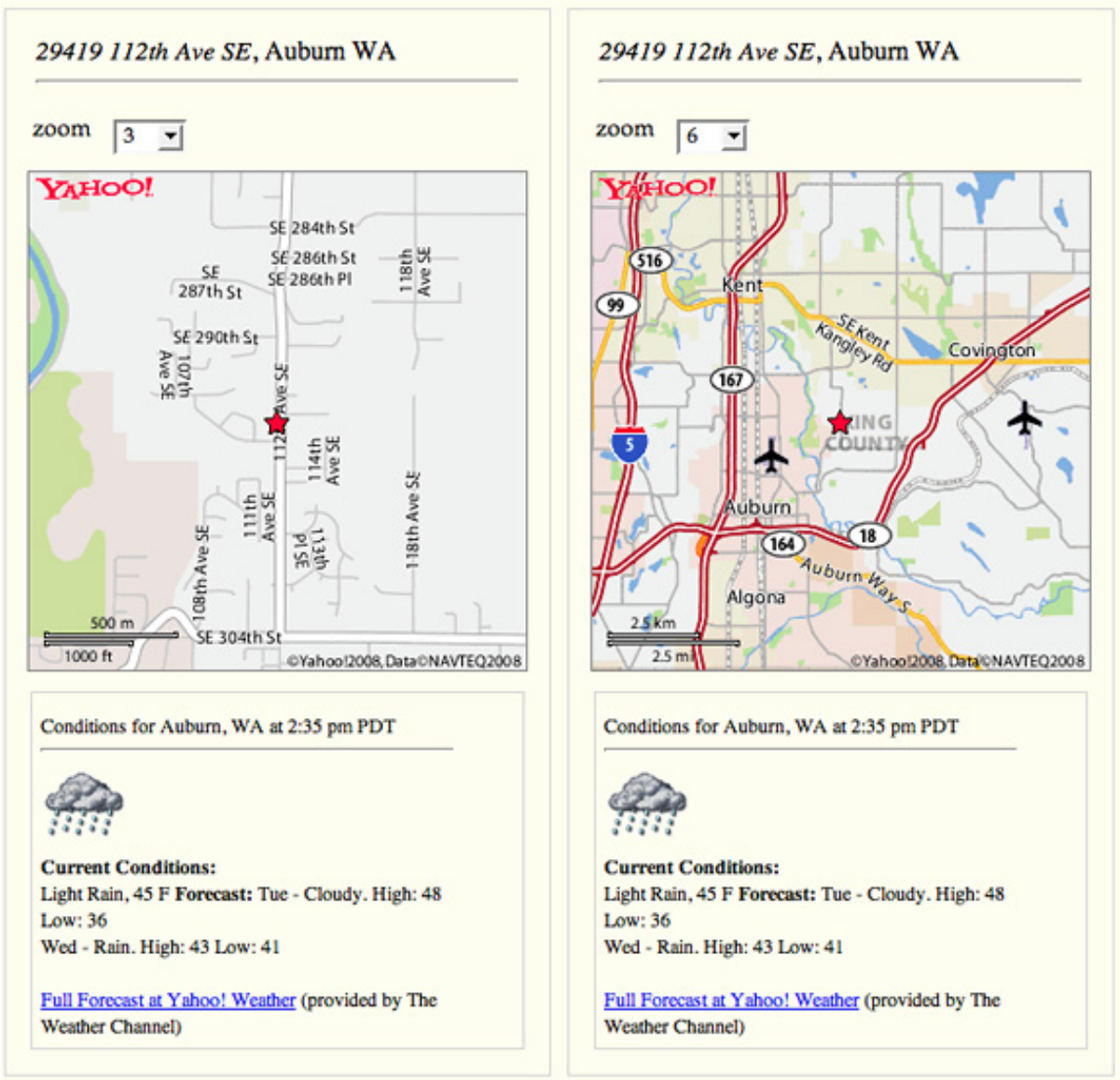


To create a place, you fill in the address form, activate the Go button, and the application passes the address to two Web services: Yahoo! Maps and Yahoo! Weather.

The Map service returns 11 map URLs pointing to maps of the address, at varying zoom levels, on the Yahoo! server. The Weather service passes back a chunk of preassembled HTML. Both image URLs and chunks of HTML are easily displayed in a JSF view, thanks to `<h:graphicImage>` and `<h:outputText>`, respectively.

The places application lets you enter as many addresses as you like. You can even use the same address more than once, as indicated by Figure 2, which is really meant to illustrate zoom levels:

Figure 2. Zoom levels



The gist of the application

The places application has four managed beans, listed in Table 1:

Table 1. The managed beans in the places application

Managed bean name	Class	Scope
`mapService`	`com.clarity.MapService`	Application
`weatherService`	`com.clarity.WeatherService`	Application
`places`	`com.clarity.Places`	Session
`place`	`com.clarity.Place`	Request

Running the places application
To run the places application, you need to get an app ID from Yahoo! at developer.yahoo.com/maps/ajax, so that you can use the Yahoo! Web services. Click on the **Get an App ID** button at the Yahoo! Maps Web Service. After you have your ID, replace `YOUR_ID_HERE` with your ID in `MapService.java` and `WeatherService.java`.

The application stores a list of `Places`, depicted in Figure 1, in session scope and maintains a `Place` in request scope. The application also provides simple APIs into Yahoo!'s map and weather Web services with the application-scoped `mapService` and `weatherService` managed beans, respectively.

Creating places is simple. Listing 1 shows the code for the address form contained in Figure 1's view:

Listing 1. The address form

```
<h:form>
  <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="2"/>
    #{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>
```

When the user activates the Go button and submits the form, JSF invokes the button's action method: `place.fetch()`. That method sends information from the Web services to `Place.addPlace()`, which creates a new `Place` instance, initializes it with the data passed to the method, and stores it in request scope.

Listing 2 shows `Place.fetch()`:

Listing 2. The Place.fetch() method

```
public class Place {
  ...
  private String[] mapUrls
  private String weather
  ...
  public String fetch() {
    FacesContext fc = FacesContext.getCurrentInstance()
    ELResolver elResolver = fc.getApplication().getELResolver()

    // Get maps

    MapService ms = elResolver.getValue(
      fc.getELContext(), null, "mapService")

    mapUrls = ms.getMap(streetAddress, city, state)
```

```
    // Get weather

    WeatherService ws = elResolver.getValue(
      fc.getELContext(), null, "weatherService")

    weather = ws.getWeatherForZip(zip, true)

    // Get places

    Places places = elResolver.getValue(
      fc.getELContext(), null, "places")

    // Add new place to places

    places.addPlace(streetAddress, city, state, mapUrls, weather)

    return null
  }
}
```

`Place.fetch()` uses JSF's variable resolver to find the `mapService` and `weatherService` managed beans, and it uses those managed beans to get map and weather information from the Yahoo! Web services. Then `fetch()` calls `places.addPlace()`, which uses the map and weather information, along with the address, to create a new `Place` in request scope.

Notice that `fetch()` returns `null`. Because `fetch()` is an action method associated with a button, that `null` return value causes JSF to reload the same view, which displays all of the places in the user's session, as shown in Listing 3:

Listing 3. Showing places in a view

```
<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">

  <h:form>
    <!-- Iterate over the list of places -->
    <ui:repeat value="#{places.placesList}" var="place">

      <div class="placeHeading">
        <h:panelGrid columns="1">

          <!-- Address at the top -->
          <h:panelGroup>
            <div style="padding-left: 5px;">
              <i><h:outputText value="#{place.streetAddress}"/></i>,
              <h:outputText value="#{place.city}"/>
              <h:outputText value="#{place.state}"/>
              <hr/>
            </div>
          </h:panelGroup>

          <!-- zoom level prompt and drop down -->
          <h:panelGrid columns="2">
            <!-- prompt -->
            <div style="padding-right: 10px;margin-bottom: 10px;font-size:14px">
              #{msgs.zoomPrompt}
            </div>
```

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

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

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

          <!-- The map -->
          <h:graphicImage url="#{place.mapUrl}" style="border: thin solid gray"/>

        </h:panelGrid>

        <!-- The weather -->
        <div class="placeMap">
          <div style="margin-top: 10px;width:250px;">
            <h:outputText style="font-size: 12px;"
              value="#{place.weather}"
              escape="false"/>
          </div>
        </div>
      </div>

    </ui:repeat>
  </h:form>

</ui:composition>
```

The code in Listing 3 uses the Facelets `<ui:repeat>` tag to loop over the list of places stored in the user's session. For each place, the output is similar to Figure 3:

Figure 3. A place shown in a view

29419 112th Ave SE, Auburn WA

zoom 5

Conditions for Auburn, WA at 6:53 pm PDT

Current Conditions:
Mostly Cloudy, 44 F **Forecast:** Tue - Showers/Wind Early. High: 48 Low: 36
Wed - Rain. High: 43 Low: 41

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

Changing zoom levels

The zoom menu (see Figure 3 and Listing 3) has an `onchange="submit()"` attribute, so the JavaScript `submit()` function submits the menu's surrounding form when the user selects a zoom level. As a result of that form submission, JSF calls the menu's associated value change listener — the `Place.zoomChanged()` method shown in Listing 4:

Listing 4. Place.zoomChanged()

```
public void zoomChanged(ValueChangeEvent e) {
  String value = e.getComponent().getValue()
  zoomIndex = (new Integer(value)).intValue()
}
```

`Place.zoomChanged()` stores the zoom level in a member variable of the `Place` class named `zoomIndex`. Because navigation is unaffected for this trip to the server, JSF reloads the same page, and the map is updated with the new zoom level, like this: `<h:graphicImage url="#{place.mapUrl}..."/>`. When the image is drawn, JSF calls `Place.getMapUrl()`, which returns the map URL for the current zoom level, as shown in Listing 5:

Listing 5. Place.getMapUrl()

```
public String getMapUrl() {
  return mapUrls == null ? "" : mapUrls[zoomIndex]
}
```

A dash of Facelets

If you've used JSF 1, you probably noticed some subtle differences in the JSF 2 code in this article. First, I'm using JSF 2's new display technology — Facelets — instead of JSP. As you will see in the subsequent articles in this series, Facelets provides many powerful features to help you implement robust, flexible, and extensible user interfaces. But in the preceding code listings, I'm not tapping into much of that power. One of the many minor improvements that Facelets brings to JSF, however, is the ability to put JSF value expressions directly into your XHTML page; for example, in Listing 1, I've put expressions such as `#{msgs.city}` directly in the page. With JSF 1, you must wrap that expression in an `<h:outputText>`, like this: `<h:outputText value="#{msgs.city}"/>`. Be aware, however, that you must always escape text that comes from user input for security reasons, so for example, in Listing 3 I use `<h:outputText>`, which escapes its text by default, to display place information.

One other thing to note, from a Facelets perspective, is the `<ui:composition>` tag in Listing 3. That tag signifies that the XHTML page in Listing 3 is meant to be included by other XHTML pages. Facelets compositions are a central component of Facelets *templating*, which is similar to the popular Apache Tiles framework. In a subsequent article in this series, I will discuss Facelets templating and show you

how to structure your views according to the *Composed Method* Smalltalk pattern.

So far, other than using Facelets, the preceding code isn't radically different from JSF 1. Now I'll show you some of the more significant differences. The first major difference is the amount of XML configuration you will write for JSF 2 applications.

Tip 1: Get rid of XML configuration

XML configuration for Web applications was always suspect — it's tedious and error prone, and it's best delegated to a framework, perhaps through annotations, conventions, or domain-specific languages. As developers, we should be able to concentrate on getting stuff done, instead of having to crank out minutiae in verbose XML.

As a case in point, Listing 6 shows the 20 lines of XML necessary for declaring managed beans in the places application with JSF 1:

Listing 6. Managed-bean declarations for JSF 1

```
<managed-bean>
  <managed-bean-class>com.clarity.MapService</managed-bean-class>
  <managed-bean-name>mapService</managed-bean-name>
  <managed-bean-scope>application</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-class>com.clarity.WeatherService</managed-bean-class>
  <managed-bean-name>weatherService</managed-bean-name>
  <managed-bean-scope>application</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-class>com.clarity.Places</managed-bean-class>
  <managed-bean-name>places</managed-bean-name>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-class>com.clarity.Place</managed-bean-class>
  <managed-bean-name>place</managed-bean-name>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

With JSF 2, the XML vanishes, and you annotate your classes instead, as shown in Listing 7:

Listing 7. Managed-bean annotations for JSF 2

```
@ManagedBean(eager=true)
public class MapService {
  ...
}

@ManagedBean(eager=true)
public class WeatherService {
  ...
}
```

```
@ManagedBean()
@SessionScoped
public class Places {
  ...
}

@ManagedBean()
@RequestScoped
public class Place {
  ...
}
```

By convention, the name of a managed bean is the same as the class name, with the first letter of the class name converted to lowercase. So, for example, the managed beans created from Listing 7, from top to bottom, are: `mapService`, `weatherService`, `places`, and `place`. You can also explicitly specify a managed bean name with the `name` attribute of the `ManagedBean` annotation, like this: `@ManagedBean(name = "place")`.

In Listing 7, I use the `eager` attribute for the `mapService` and `webService` managed beans. When the `eager` attribute is `true`, JSF creates the managed bean at startup and places it in application scope.

You can also set managed bean properties with the `@ManagedProperty` annotation. Table 2 shows the complete list of JSF 2 managed bean annotations:

Table 2. JSF 2 managed bean annotations (@...Scoped annotations are only valid with @ManagedBean)

Managed-bean annotation	Description	Attributes
`@ManagedBean`	Registers an instance of this class as a managed bean and places it in the scope specified with one of the `@...Scoped` annotations. If no scope is specified, JSF puts the bean in request scope, and if no name is specified, JSF converts the first letter in the classname to lowercase to come up with a managed bean name; for example, if the classname is `UserBean`, JSF creates a managed bean named `userBean`. Both the `eager` and `name` attributes are optional. This annotation must be used with a Java class that implements a zero-argument constructor.	`eager`, `name`

`@ManagedProperty`	Sets a property of a managed bean. The annotation must be placed before the declaration of the class's member variable. The `name` attribute specifies the name of the property, which defaults to the name of the member variable. The `value` attribute is the value of the property and can be either a string or a JSF expression, such as `#{...}`.	`value`, `name`
`@ApplicationScoped`	Stores the managed bean in application scope.	
`@SessionScoped`	Stores the managed bean in session scope.	
`@RequestScoped`	Stores the managed bean in request scope.	
`@ViewScoped`	Stores the managed bean in view scope.	
`@NoneScoped`	Specifies that the managed bean has no scope. Managed beans with no scope are useful when they are referenced by other beans.	
`@CustomScoped`	Stores the managed bean in a custom scope. A custom scope is simply a map that's accessible to page authors. You can programmatically control the visibility and lifetime of beans in custom scopes. The `value` attribute points to a map.	`value`

Removing managed bean declarations from faces-config.xml significantly reduces your XML, but you can get rid of virtually all of it with JSF 2 through either annotations, as I'm doing for managed beans, or with convention, as is the case for JSF 2's simplified navigation handling.

Tip 2: Simplify navigation

In JSF 1, navigation was specified in XML. For example, to go from login.xhtml to places.xhtml, you might use the navigation rule in Listing 8:

Listing 8. Navigation configuration rules and cases for JSF 1

```
<navigation-rule>
  <navigation-case>
    <from-view-id>/pages/login.xhtml</from-view-id>
    <outcome>places</outcome>
    <to-view-id>/pages/places.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

To get rid of the XML in Listing 8, you can take advantage of JSF 2's navigation convention: JSF adds .xhtml to the end of a button's action and loads that file. That means you don't need annotations or anything other than convention to get out of the navigation-rule-writing business entirely. In Listing 9, the button's action is `places`, so JSF loads places.xhtml:

Listing 9. Navigation by convention

```
<h:commandButton id="loginButton"
  value="#{msgs.loginButtonText}"
  action="places"/>
```

No navigation XML is required for Listing 9. The button in Listing 9 loads places.xhtml, but only if the file is in the same directory as the file in which the button resides. If the action doesn't begin with a forward slash (`/`), JSF assumes that it's a relative path. If you want to be more explicit, you can specify an absolute path, as shown in Listing 10:

Listing 10. Navigation with absolute paths

```
<h:commandButton id="loginButton"
  value="#{msgs.loginButtonText}"
  action="/pages/places"/>
```

When the user activates the button in Listing 10, JSF loads the /pages/places.xhtml file.

By default, JSF forwards from one XHTML page to another, but you can redirect instead by specifying the `faces-redirect` parameter, as illustrated in Listing 11:

Listing 11. Navigation by redirect

```
<h:commandButton id="loginButton"
  value="#{msgs.loginButtonText}"
  action="places?faces-redirect=true"/>
```

Tip 3: Use Groovy

The best thing about Java technology is not the Java language, but the Java virtual machine (JVM). Powerful, new, and innovative languages such as Scala, JRuby, and Groovy run on the JVM, giving you other dimensions in which to write code.

Groovy — an ineptly named but highly capable blend of Ruby, Smalltalk, and the Java language — is one of the more popular of these languages (see Resources).

There are many reasons to use Groovy, starting with the fact that it is much less verbose and much more powerful than its second cousin, the Java language. Two more reasons: no semicolons and no casting required.

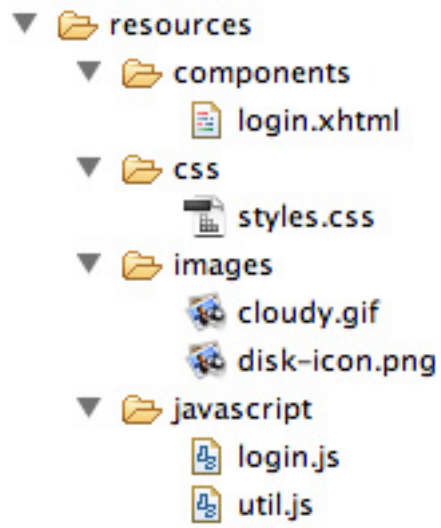
You may not have noticed, but Listing 2, for the `Place` class, is written in Groovy. The lack of semicolons is a subtle clue, but also notice this line of code: `MapService ms = elResolver.getValue(...)`. With Java code, I would have to cast the result of `ElResolver.getValue()`, because that method returns type `Object`. Groovy does the cast for me.

You can use Groovy for any JSF artifacts that you might normally write in Java code — for example, components, renderers, validators, and converters. In fact, this is nothing new to JSF 2 — because Groovy source files compile to Java bytecode, you can just use the Groovy-generated .class files as though they were generated by `javac`. Of course, as soon as you've got that working, you'll want to know how to hot-deploy Groovy source code, and for Eclipse users, the answer is simple: download and install the Groovy Eclipse plug-in (see Resources). Mojarra, which is Sun's JSF implementation, has had explicit support for Groovy since version 1.2_09 (see Resources).

Tip 4: Use resource handlers

JSF 2 provides a standard mechanism for defining and accessing resources. You put your resources under a top-level directory named resources, and use some JSF 2 tags to access those resources in your views. For example, Figure 4 shows the resources for the places application:

Figure 4. The places application's resources



The only requirement for a resource is that it reside in the resources directory or a subdirectory thereof. You can name subdirectories of the resources directory anything you want.

In your view code, you can access resources with a couple of JSF 2 tags: `<h:outputScript>` and `<h:outputStylesheet>`. Those tags work in concert with JSF 2's `<h:head>` and `<h:body>` tags, as shown in Listing 12:

Listing 12. Accessing resources in XHTML

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

  <h:head>
    ...
  </h:head>

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

The `<h:outputScript>` and `<h:outputStylesheet>` tags have two attributes that identify the script or stylesheet, respectively: `library` and `name`. The `library` name corresponds to the directory, under the resources directory, in which the resource resides. For example, if you have a stylesheet in a resources/css/en directory, the `library` would be `css/en`. The `name` attribute is the name of the resource itself.

Relocatable resources

It's important that developers can specify where in a page they want their resource to appear. For example, if you put JavaScript in the body of a page, the browser will

execute the JavaScript when the page loads. On the other hand, if you place JavaScript in the head of a page, that JavaScript will only be executed when called. Because a resource's location can affect how it's used, you need to be able to specify where you want resources to end up.

JSF 2 resources are *relocatable*, meaning you can specify the location in the page where you want them placed. You specify that location with the `target` attribute; for example, in Listing 12, I put CSS in the body and JavaScript in the head.

Sometimes you'll need to access a resource using the JSF expression language (EL). For example, Listing 13 shows how you can access an image with `<h:graphicImage>`:

Listing 13. Accessing resources with the JSF expression language

```
<h:graphicImage value="#{resource['images:cloudy.gif']}"/>
```

An non-EL alternative to Listing 13

Admittedly, the syntax in Listing 13 is somewhat counterintuitive. It actually accesses a map created by JSF to store resources, so you rarely need to use that syntax. In fact, you can access images with `<h:graphicImage/>` without using the EL, like this:

```
<h:graphicImage library="images"
name="cloudy.gif"/>
```

The syntax for accessing resources within an EL expression is `resource['`*`LIBRARY`*`:`*`NAME`*`']`, where *`LIBRARY`* and *`NAME`* correspond to the `library` and `name` attributes of the `<h:outputScript>` and `<h:outputStylesheet>` tags.

Still to come

I've barely scratched the surface of JSF 2 features so far, with managed bean annotations, simplified navigation, and support for resources. In the remaining two articles in this series, I will explore Facelets, JSF 2's composite components, and built-in support for Ajax.

Downloads

Description	Name	Size	Download method
Source code	jsf2fu1.zip	1.9MB	HTTP

Information about download methods

Resources

Learn

- The JSF home page: Find more resources about developing with JSF.
- "Groovy+Mojarra" (Ryan Lubke's Blog, April 2008): Lubke, a Sun engineer discusses using Groovy with Mojarra 1.2_09.
- "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.
- 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.
- *Practically Groovy* (Andrew Glover and Scott Davis, developerWorks): Get up to speed with Groovy.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- JSF: Download JSF 2.0.
- Groovy Eclipse Plugin: Use this plug-in to simplify Groovy development.

Discuss

- Check out developerWorks blogs and 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 speaks regularly at conferences and user groups. He has been a

regular on the NFJS tour since 2003, has taught courses at Java University, 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.