1. **JSF 2.0 features vs JSF 1.2**
2. JSP 2.x to Facelets 2.0 changing procedure
3. **Annotations of JSF**

### POST-REDIRECT-GET and JSF 2.0

**Decisive**निर्णायक Nerd बेवकूफ

**Below is the list of JSF 2.0 features vs JSF 1.2**

* **AJAX support within tags** – had to write bespoke tags previously
* **Composite components** – This feature enables creating true JSF components as aggregations of other JSF components. With this feature, it is easy to build your own custom JSF components and to refactor your existing views into reusable components, complete with listeners, attributes, and events.
* **Partial State Saving** – one of the biggest complaints agasint JSF has been the amount of memory it consumes to maintain view state between requests. This feature dramatically reduces the amount of memory for this purpose, as well as greatly simplifying the API for handling state when developing custom components.
* **View Parameters** – Another big complaint against JSF is insistence on using POST for inter-page navigations.
* **Navigation Enhancements** – JSF 2.0 bring several enhancements to navigation, including bookmarkability, navigation with XML navigation rules, conditional navigation, support for the POST-REDIRECT-GET pattern, flash, and runtime, and runtime inspection of navigation rules.
* **Exception Handling** – JSF 2.0 now has a central ExceptionHandler through which all exception funnelled. This enables easily constructing an error page that uses JSF components.
* **Expression Language Enhancements** – Several new implicit objects have been introduced, and the EL now supports method invocation on arbitrary Java methods, including parameter passing.
* **Validation –** An entirely new Java specification has been developed to address validation, JSR-303 Bean Validation. This specification integrates well with JSF 2.0.
* **New scopes –** In addition to the flash scope, JSF 2.0 also provides a mechanism for defining custom scopes.
* **FacesContext Servlet** access is now available during application startup and shutdown.
* **JSF 2.0** now allows the developer to tell the runtime in what phase of the software development lifecycle the runtime is executing.
* **Annotations** may obviate the need for XML. Navigation can be completely be done without XML configuration

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Painfulness**  upgrading   * JSP 2.x to JSP 2.x = Almost no effort. * Facelets 1.x to Facelets 2.0 = Little effort. * JSP 2.x to Facelets 2.0 = Lot of effort. Double this if you also have custom components.   **Basic changes**  Regardless of the view technology switch, *at least* the following steps should be done:   * Remove JSF 1.2 JAR's from /WEB-INF/lib (if any). * Drop JSF 2.0 JAR's in /WEB-INF/lib (if JSF 1.2 was servletcontainer-supplied, you might want to change the classloading policy to load webapp libraries first before servletcontainer libraries, see also [JSF2 classloading issues in application servers](http://stackoverflow.com/questions/5815623/jsf-2-issues-in-application-servers)). * Update root declaration of faces-config.xml to comply JSF 2.0 spec. * <faces-config * xmlns="http://java.sun.com/xml/ns/javaee" * xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" * xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-facesconfig\_2\_0.xsd"   version="2.0">   * Ensure that root declaration of web.xml already complies *at least* Servlet 2.5. JSF 2.0 won't work on 2.4 or lower ([although it's hackable](http://stackoverflow.com/questions/5998447/running-jsf-2-0-on-servlet-2-4-container)). * <web-app * xmlns="http://java.sun.com/xml/ns/javaee" * xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" * xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" * xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" * id="YourWebappID"   version="2.5">  **JSP 2.x to Facelets 2.0**  If you're using **JSP 2.x** as view technology and you want to upgrade to **Facelets 2.0** immediately, then you need to do a lot of changes before the site can go live. You're basically changing the view technology here.  **Master page changes**  On every master page, you need to change the following basic JSP template..  <%@page contentType="text/html" pageEncoding="UTF-8"%>  <%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>  <%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>  <!DOCTYPE html>  <f:view>  <html lang="en">  <head>  <title>JSP page</title>  </head>  <body>  <h:outputText value="JSF components here." />  </body>  </html>  </f:view>  ..to the following basic Facelets template:  <!DOCTYPE html>  <html lang="en"  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:head>  <title>XHTML page</title>  </h:head>  <h:body>  <h:outputText value="JSF components here." />  </h:body>  </html>  **Include page changes**  If your existing JSP pages are well designed, you should not have any line of *scriptlet* code and you should also have only the <jsp:include> as the sole JSP-specific tag. Any of those needs to be changed from:  <jsp:include page="include.jsp" />  to  <ui:include src="include.xhtml" />  The basic JSP include page template of..  <%@page contentType="text/html" pageEncoding="UTF-8"%>  <%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>  <%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>  <f:subview id="include">  <h:outputText value="JSF components here." />  </f:subview>  ..should be changed to the following basic Facelets include page template:  <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:outputText value="JSF components here." />  </ui:composition>  **Custom component changes**  You need to change the JSP TLD files to Facelets TLD files as described in this [Mojarra Migration Guide](http://javaserverfaces.java.net/nonav/rlnotes/2.0.0/migration.html).  **Annotations of JSF**  Regardless of the migration approach, you can gradually [eliminate](http://blogs.oracle.com/rlubke/entry/faces_config_xml_we_don) the faces-config.xml by the new JSF 2.0 annotations. Any <managed-bean> can be annotated by [@ManagedBean](http://download.oracle.com/javaee/6/api/javax/faces/bean/ManagedBean.html):  @ManagedBean(name="managedBeanName")  @RequestScoped  public class SomeBean {}  Next to [@RequestScoped](http://download.oracle.com/javaee/6/api/javax/faces/bean/RequestScoped.html), there are also [@ViewScoped](http://download.oracle.com/javaee/6/api/javax/faces/bean/ViewScoped.html), [@SessionScoped](http://download.oracle.com/javaee/6/api/javax/faces/bean/SessionScoped.html) and [@ApplicationScoped](http://download.oracle.com/javaee/6/api/javax/faces/bean/ApplicationScoped.html) available. If you omit the name attribute of the @ManagedBean, then it will default to classname with the 1st char lowercased.  @ManagedBean  @RequestScoped  public class SomeBean {}  In this particular example, it will be #{someBean}.  Any <managed-property> can be annotated using [@ManagedProperty](http://download.oracle.com/javaee/6/api/javax/faces/bean/ManagedProperty.html):  @ManagedProperty("#{otherBean}")  private OtherBean otherBean;  Any <validator> can be annotated using [@FacesValidator](http://download.oracle.com/javaee/6/api/javax/faces/validator/FacesValidator.html):  @FacesValidator("someValidator")  public class SomeValidator implements Validator {}  Any <converter> can be annotated using [@FacesConverter](http://download.oracle.com/javaee/6/api/javax/faces/convert/FacesConverter.html)  @FacesConverter("someConverter")  public class SomeConverter implements Converter {}  Any <renderer> can be annotated using [@FacesRenderer](http://download.oracle.com/javaee/6/api/javax/faces/render/FacesRenderer.html)  @FacesRenderer(componentFamily="someComponentFamily", rendererType="someRendererType")  public class SomeRenderer extends Renderer {}  Any <navigation-case> which uses the filename of the XHTML page as both <from-outcome> and <to-view-id> can be removed since this will be [implicitly](http://andyschwartz.wordpress.com/2009/07/31/whats-new-in-jsf-2/#navigation-implicit) done. This can be gradually done by changing all outcome values to match the filename of the target view.  Finally, any session scoped bean which was been put in the session with the sole reason to retain the bean data in subsequent requests in the same tab/window can better be marked @ViewScoped, because this way the bean won't be affected when the enduser opens the same page in different tabs/windows.  **Component libraries**  Note that I don't take any 3rd party componant libraries like PrimeFaces/RichFaces/IceFaces into account in this answer, it would then be impossible to write a reliable answer since it basically boils down to "it depends". In general it's sufficient to just upgrade the component library to a -by themselves verified- JSF 2.0 compatible version as per their instructions. Best is to just write unit tests, run them before and after the upgrade and fix any issues individually.  Here are at least some useful links with regard to migration of the specific component library:   * [RichFaces Migration Guide - 3.3.x to 4.x migration](http://community.jboss.org/wiki/RichFacesMigrationGuide33x-4xMigration) * [IceFaces 2 Wiki - IceFaces 1.x Compatibility Guide](http://wiki.icefaces.org/display/ICE/ICEfaces+1.x+Compatibility)   PrimeFaces has no migration guide for PrimeFaces 1.x to 2.x as PrimeFaces 1.x requires Facelets 1.x already, so you just have to follow Facelets 1.x to 2.x migration steps. However, there's a PrimeFaces [2.x to 3.x migration guide](http://code.google.com/p/primefaces/wiki/MigrationGuideFrom22to30) which might apply as well on migrating from PrimeFaces 1.x to 3.x. Tomahawk has also no migration guide. Basically the only which you need to change are the JARs and if necessary get rid of all <t:saveState> references on a request scoped bean by making the bean view scoped.   |  |  |  | | --- | --- | --- | | [share](http://stackoverflow.com/a/4532870)|[improve this answer](http://stackoverflow.com/posts/4532870/edit) | [edited Jul 27 '12 at 22:40](http://stackoverflow.com/posts/4532870/revisions)  [Bill the Lizard](http://stackoverflow.com/users/1288/bill-the-lizard)♦ 160k107388669 | answered Dec 26 '10 at 5:27  [[https://www.gravatar.com/avatar/89927e2f4bde24991649b353a37678b9?s=32&d=identicon&r=PG](http://stackoverflow.com/users/157882/balusc)](http://stackoverflow.com/users/157882/balusc)  [BalusC](http://stackoverflow.com/users/157882/balusc) 531k12216271944 | |
|  | |  |  |  |  | | --- | --- | --- | --- | | |  |  | | --- | --- | |  |  | | @ManagedBean(name="managedBeanName") @RequestScoped It is :) –  [Daniel Szalay](http://stackoverflow.com/users/157762/daniel-szalay)[Jan 29 '11 at 10:25](http://stackoverflow.com/questions/4441713/migrating-from-jsf-1-2-to-jsf-2-0/#comment5368007_4532870) | | |  |  | | --- | --- | |  |  | | great post, helped me a lot. Something to note: when moving from jsf 1.2 to jsf 2 you can be almost certain people have used a4j from richfaces 3.3.x. I decided to use richfaces 3.3.3 together with jsf 2 since it seemed like a mediocre change to perform to upgrade to richfaces 4.x . So I did your guide (undid all facelets related things in faces-config (activated viewhandler removed taglig annotation), then followed [community.jboss.org/wiki/RichFaces333AndJSF20](http://community.jboss.org/wiki/RichFaces333AndJSF20) and finally had to do this [stackoverflow.com/questions/85532/…](http://stackoverflow.com/questions/85532/faces-servlet-threw-exception-java-lang-stackoverflowerror) –  [Toskan](http://stackoverflow.com/users/533426/toskan)[Dec 29 '11 at 19:19](http://stackoverflow.com/questions/4441713/migrating-from-jsf-1-2-to-jsf-2-0/#comment10779480_4532870) | | |  |  | | --- | --- | |  |  | | Great answer. In my case I also had to set the javax.faces.VALIDATE\_EMPTY\_FIELDS parameter to false to get the validation sorted. See also: [stackoverflow.com/questions/6113935/…](http://stackoverflow.com/questions/6113935/shouldnt-the-validation-be-skipped-when-there-is-no-value-specified) –  [Jasper de Vries](http://stackoverflow.com/users/880619/jasper-de-vries)[Jul 9 at 13:07](http://stackoverflow.com/questions/4441713/migrating-from-jsf-1-2-to-jsf-2-0/#comment38218267_4532870) | | |  |  | | --- | --- | | 1 |  | | I can also recommend everybody to read [balusc.blogspot.nl/2011/09/communication-in-jsf-20.html](http://balusc.blogspot.nl/2011/09/communication-in-jsf-20.html) –  [Jasper de Vries](http://stackoverflow.com/users/880619/jasper-de-vries)[Jul 11 at 9:33](http://stackoverflow.com/questions/4441713/migrating-from-jsf-1-2-to-jsf-2-0/#comment38292293_4532870) |   add a comment |

* JSF - Java Server Faces is the overarching application development framework. It contains many aspects, one of which is the view handler. <http://en.wikipedia.org/wiki/JavaServer_Faces>
* Facelets is a view handler for JSF. Before JSF 2.0, JSP was the default view handler, the problem was, JSP didn't work very well with JSF's component based framework. Facelets was designed from the ground up to work well with JSF. <http://en.wikipedia.org/wiki/Facelets>

Tag libraries are another aspect to JSF, but Facelets is not a taglib, and many tag libs can be used with Facelets.

### POST-REDIRECT-GET and JSF 2.0

All interactive programs provide two basic functions: obtaining user input and displaying the results. Web applications implement this behavior using two HTTP methods: POST and GET respectively. This simple protocol gets broken when an application returns a web page in response to a POST request. Peculiarities of the POST method combined with idiosyncrasies of different browsers often lead to an unpleasant user experience and may produce an incorrect state of the server application.

To address the problem, Jouravlev described a technique that he called POST-REDIRECT-GET, or the PRG pattern for short. The rules of the pattern are as follows:

* Never show pages in response to POST
* Always load pages using GET
* Navigate from POST to GET using REDIRECT

Previous versions of [JavaServer Faces (JSF) technology](http://java.sun.com/javaee/javaserverfaces/) violated the first of these rules by using POST for every page navigation. In navigating from one page to another in a JSF-enabled application, the JSF framework forwarded a POST request through the Servlet API's RequestDispatcher.forward( ) method. This caused a new Faces page to be rendered and returned to the browser in response to the postback request.

Indeed, most popular Java Servlet-based web frameworks, including Struts, use this approach for navigation. HTTP purists rightly point out that this approach violates the first rule in the PRG pattern. Not only did JSF violate the first rule, but until [JavaServer Faces 2.0](http://jcp.org/en/jsr/detail?id=314), it was very difficult to do it any other way. Thanks to a JSF contribution from the Seam team at JBoss, it is now much easier to do PRG with JSF.

This Tech Tip shows how to implement the PRG pattern in JSF 2.0. The content of the tip is an adaptation of a section on PRG and JSF 2.0 in my upcoming book, with Neil Griffin, [JavaServer Faces 2.0: The Complete Reference](http://bit.ly/5qrXJ).

**A Non-PRG Example**

Let's start by examining a simple JSF 2.0 application that handles user registration. In this first example, the application does not implement the PRG pattern. The initial page for the application is coded in file register.xhtml, as follows:

<!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:f="http://java.sun.com/jsf/core">

<h:head>

<title>A Simple JavaServer Faces Registration Application</title>

</h:head>

<h:body>

<h:form>

<h2>JSF Registration App</h2>

<h4>Registration Form</h4>

<table>

<tr>

<td>First Name:</td>

<td>

<h:inputText label="First Name"

id="fname" value="#{userBean.firstName}"

required="true"/>

<h:message for="fname" />

</td>

</tr>

<tr>

<td>Last Name:</td>

<td>

<h:inputText label="Last Name"

id="lname" value="#{userBean.lastName}"

required="true"/>

<h:message for="lname" />

</td>

</tr>

... additional table rows not shown.

</table>

<p><h:commandButton value="Register" action="confirm" /></p>

</h:form>

</h:body>

</html>

|  |
| --- |
|  |

The page presents text fields for the user to enter a first name and a last name. It also displays a Register button. When the user presses the Register button, the JSF navigation rule system looks for a page within the application whose extension is the same as the current page and whose filename is confirm. If confirm.xhtml exists, JSF uses the navigation components in that file to navigate to the next page. Here is the confirm.xhtml file:

<!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:f="http://java.sun.com/jsf/core">

<h:head>

<title>A Simple JavaServer Faces Registration Application</title>

</h:head>

<h:body>

<h:form>

<h2>JSF Registration App</h2>

<h4>Registration Confirmation</h4>

<table>

<tr>

<td>First Name:</td>

<td>

<h:outputText value="First Name" value="#{userBean.firstName}"

</td>

</tr>

<tr>

<td>Last Name:</td>

<td>

<h:outputText label="Last Name" value="#{userBean.lastName}"

</td>

</tr>

... additional table rows not shown.

</table>

<p><h:commandButton value="Edit" action="register" /></p>

<p><h:commandButton value="Confirm" action="#{userBean.addConfirmedUser}" /></p>

</h:form>

</h:body>

</html>

|  |
| --- |
|  |

The confirm.xhtml file includes markup for an Edit button and a Confirm button. If the user clicks the Edit button, he or she is taken back to the register.xhtml page. If the user clicks the Confirm button, an action is invoked. The Confirm button specifies an action method, addConfirmedUser( ), that determines the outcome programmatically in the logic of the method. Here is the UserBean.java file, which contains the addConfirmedUser( ) method:

package com.jsfcompref.model;

... imports

@ManagedBean

@SessionScoped

public class UserBean {

... properties and methods

public String addConfirmedUser() {

boolean added = true; // actual application may fail to add user

FacesMessage doneMessage = null;

String outcome = null;

if (added) {

doneMessage = new FacesMessage("Successfully added new user");

outcome = "done";

} else {

doneMessage = new FacesMessage("Failed to add new user");

outcome = "register";

}

FacesContext.getCurrentInstance().addMessage(null, doneMessage);

return outcome;

}

|  |
| --- |
|  |

For this simple case, addConfirmedUser( ) causes a message stating Successfully added new user to be displayed on the page and returns "done" as the outcome. When the addConfirmedUser( ) method returns "done" as the outcome, it takes the user to the done.xhtml page. This is an example of *implicit navigation*, a new feature in JSF 2.0. If no matching navigation case is found after checking all available rules, the navigation handler checks to see whether the action outcome corresponds to a view id. If a view matching the action outcome is found, an implicit navigation to the matching view occurs. Here the outcome is "done" and the matching view is done.xhtml, so the user is taken to the done.xhtml page. Implicit navigation saves you the effort of adding navigation rules in the faces-config.xml file.

Here is the done.xhtml page:

<!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:f="http://java.sun.com/jsf/core">

<h:head>

<title>A Simple JavaServer Faces Registration Application</title>

</h:head>

<h:body>

<h:form>

<h2>JSF Registration App</h2>

<h4>Registration Confirmation</h4>

<h:messages />

<table>

<tr>

<td>First Name:</td>

<td>

<h:outputText value="First Name" value="#{userBean.firstName}"

</td>

</tr>

</table>

</h:form>

</h:body>

</html>

|  |
| --- |
|  |

**POST-REDIRECT-GET Using View Parameters**

*View Parameters* is a simple, declarative way to map incoming request parameter values to special components within the view. These mappings are specified using the new <f:viewParam> component, within the new <f:metadata> section of the view. Consider the following example:

<f:metadata>

<f:viewParam name="foo" value="#{bean.foo}"/>

</f:metadata>

|  |
| --- |
|  |

This example specifies that the value of the request parameter with the name "foo" is automatically assigned to the property at #{bean.foo}. So for a GET request as follows:

page1.jspx?foo=bar

|  |
| --- |
|  |

The value of the #{bean.foo} property will be set to bar when JSF starts processing the request.

View Parameters is similar in spirit to the page parameters feature found in JBoss Seam, but the JSF 2.0 incarnation of the feature is tightly integrated with the core JSF specification, making the feature easier to use and more powerful. Let’s look at another simple example.

<!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:f="http://java.sun.com/jsf/core">

<h:head>

<title>A Simple JavaServer Faces 2.0 View</title>

</h:head>

<h:body>

<h:form>

<p>First Name:<<h:inputText id="fname"

value="#{userBean.firstName}" /></p>

<p><h:commandButton value="submit"

action="page02?faces-redirect=true**&amp;includeViewParams=true"** /></p>

</h:form>

</h:body>

</html>

|  |
| --- |
|  |

The <h:commandButton> element has action="page02?faces-redirect=true". In the Internet standard that defines URLs, the presence of a ? character indicates the remainder of the URL will be an & or &amp;-separated list of name=value pairs that should be submitted to the server along with the request for the URL. This is known as a *query string*. JSF borrows the meaning of the ? character here, and the meaning is exactly the same as in the Internet standard for URLs. There are two special query strings parameters recognized by JSF when it parses the outcome on the server side. The faces-redirectquery string tells the navigation system that this implicit navigation case must be treated as if it were a real <navigation-case> element that includes a <redirect/> element. The other special query string parameter, includeViewParams, tells the navigation handler to include the view parameters when performing the navigation. But what view parameters should be included? The view parameters to be included when performing the navigation are declared on the to-view-id page. In this case, we are using implicit navigation, so the implicit to-view-id is page02.xhtml, shown below.

<!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:f="http://java.sun.com/jsf/core">

**<f:metadata>**

**<f:viewParam name="fname> value="#userBean.firstName}"/>**

**</f:metadata>**

<h:head>

<title>A Simple JavaServer Faces 2.0 View</title>

</h:head>

<h:body>

<h:form>

<p> Hello #{userBean.firstName}.</p>

</h:form>

</h:body>

</html>

|  |
| --- |
|  |

When the navigation handler encounters the matching navigation-case (implicit or explicit) that declares that view parameters should be included, it looks at the view parameters of the from-view-id and to-view-id pages and performs a match-and-copy algorithm to convey the view parameters to the new page. In this case, the navigation-case also requested a redirect.

Now let’s look at the registration example, this time implemented to do PRG with view parameters. The register.xhtml page looks like this:

<!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:f="http://java.sun.com/jsf/core">

**<f:metadata>**

**<f:viewParam name="fname" value="#{userBean.firstName}" />**

**<f:viewParam name="lname" value="#{userBean.lastName}" />**

**<f:viewParam name="sex" value="#{userBean.sex}" />**

**<f:viewParam name="dob" value="#{userBean.dob}">**

**<f:convertDateTime pattern="MM-dd-yy" />**

**</f:viewParam>**

**<f:viewParam name="mail" value="#{userBean.email}" />**

**<f:viewParam name="sLevel" value="#{userBean.serviceLevel}" />**

**</f:metadata>**

<h:head>

<title>A Simple JavaServer Faces Registration Application</title>

</h:head>

<h:body>

<h:form>

<h2>JSF Registration App</h2>

<h4>Registration Form</h4>

<table>

<tr>

<td>First Name:</td>

<td>

<h:inputText label="First Name"

id="fname" value="#{userBean.firstName}"

required="true"/>

<h:message for="fname" />

</td>

</tr>

... remaining table rows omitted, they are the same as the original

</table>

<!-- The query parameters on the action attribute cause JSF to do the

POST REDIRECT GET pattern -->

<p><h:commandButton value="Register"

action=**"confirm?faces-redirect=true&amp;includeViewParams=true"** /></p>

</h:form>

</h:body>

</html>

|  |
| --- |
|  |

In the previous View Parameters example, we stated that <code.<f:viewparam>< code=""> elements only appear on the to-view-id page. That is still true in this example, even though this is the first page the user sees. This particular application allows the user to go back and forth between the register.xhtml page and the confirm.xhtml page. Therefore, when the user is on the confirm.xhtml page, the to-view-id is the register.xhtml page and vice versa. Thus, <f:viewParams> is on both pages. You will need an <f:viewParam> for every input component on the from page that you wish to carry forward to the to page. Also note the <f:convertDateTime> within the <f:viewParam> for the dob property. This is necessary because, when the navigation is performed, the converter needs to be invoked to carry the value forward. If there is an explicit converter defined on the input field, then there must also be one on the corresponding <f:viewParam>. Finally, you can see the now familiar extra query parameters on the implicit navigation: confirm?faces-redirect=true&amp;includeViewParams=true. </code.<f:viewparam><>

Let’s examine the changes to the UserBean.java file.

package com.jsfcompref.model;

... omits imports

@ManagedBean

**@RequestScoped**

public class UserBean {

... additional properties omitted

public String addConfirmedUser() {

boolean added = true; // actual application may fail to add user

FacesMessage doneMessage = null;

String outcome = null;

if (added) {

doneMessage = new FacesMessage("Successfully added new user");

outcome = **"done?faces-redirect=true&amp;includeViewParams=true";**

} else {

doneMessage = new FacesMessage("Failed to add new user");

outcome = **"register?faces-redirect=true&amp;includeViewParams=true";**

}

FacesContext.getCurrentInstance().addMessage(null, doneMessage);

return outcome;

}

|  |
| --- |
|  |

The only changes to this code are to make the bean be request-scoped and to add the query parameters to the implicit navigation string. In this case, the includeViewParams=true parameter is added, causing whatever view parameters declared on the to-view-id page to be included in the navigation.

The confirm.xhtml page follows:

<!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:f="http://java.sun.com/jsf/core">

**<f:metadata>**

**<f:viewParam name="fname" value="#{userBean.firstName}" />**

**<f:viewParam name="lname" value="#{userBean.lastName}" />**

**<f:viewParam name="sex" value="#{userBean.sex}" />**

**<f:viewParam name="dob" value="#{userBean.dob}">**

**<f:convertDateTime pattern="MM-dd-yy" />**

**</f:viewParam>**

**<f:viewParam name="mail" value="#{userBean.email}" />**

**<f:viewParam name="sLevel" value="#{userBean.serviceLevel}" />**

**</f:metadata>**

<h:head>

<title>A Simple JavaServer Faces Registration Application</title>

</h:head>

<h:body>

<h:form>

<h2>JSF Registration App</h2>

<h4>Registration Confirmation</h4>

<table>

<tr>

<td>First Name:</td>

<td>

<h:outputText value="First Name" value="#{userBean.firstName}"

</td>

</tr>

... additional rows omitted, they are the same as the original.

</table>

<p><h:commandButton value="Edit"

action=**"register?faces-redirect=true&amp;includeViewParams=true"** /></p>

</h:form>

**<h:form>**

**<h:inputHidden value="#{userBean.firstName}" />**

**<h:inputHidden value="#{userBean.lastName}"/>**

**<h:inputHidden value="#{userBean.sex}" />**

**<h:inputHidden value="#{userBean.dob}">**

**<f:convertDateTime pattern="MM-dd-yy" />**

**</h:inputHidden>**

**<h:inputHidden value="#{userBean.email}" />**

**<h:inputHidden value="#{userBean.serviceLevel}" />**

**<p><h:commandButton value="Confirm"**

**action="#{userBean.addConfirmedUser}" /></p>**

**</h:form>**

</h:body>

</html>

|  |
| --- |
|  |

As in the register.xhtml page, we need the <f:metadata> section at the top of the page and the additional query parameters on the action string. What is new here are the additional <h:form> element and <h:inputHidden> elements, and the fact that the Confirm button has been moved into this new form. This is necessary because we need to carry forward to the next page as regular form submit parameters the values passed to this page as view parameters. But there are no regular input fields as there are on the register.xhtml page. Therefore, we use hidden fields to carry the values forward. Note also the continued necessity for the <f:convertDateTime> on the dob field.

Finally, here is the done.xhtml page:

<!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:f="http://java.sun.com/jsf/core">

**<f:metadata>**

**<f:viewParam name="fname" value="#{userBean.firstName}" />**

**<f:viewParam name="lname" value="#{userBean.lastName}" />**

**<f:viewParam name="sex" value="#{userBean.sex}" />**

**<f:viewParam name="dob" value="#{userBean.dob}">**

**<f:convertDateTime pattern="MM-dd-yy" />**

**</f:viewParam>**

**<f:viewParam name="email" value="#{userBean.email}" />**

**<f:viewParam name="sLevel" value="#{userBean.serviceLevel}" />**

**</f:metadata>**

<h:head>

<title>A Simple JavaServer Faces Registration Application</title>

</h:head>

<h:body>

<h:form>

<h2>JSF Registration App</h2>

<h4>Registration Confirmation</h4>

<h:messages />

<table>

<tr>

<td>First Name:</td>

<td>

<h:outputText value="First Name" value="#{userBean.firstName}"

</td>

</tr>

... additional rows omitted

</table>

</h:form>

</h:body>

</html>

|  |
| --- |
|  |

The only difference between this done.xhtml and the original one is the now familiar <f:metadata> section.

**Running the Sample Code**

A sample application that implements PRG accompanies this tip. These instructions use the Maven 2 software project management tool to build the sample application and then deploy it in the GlassFish v3 Preview application server.

1. If you haven't already done so, download a recent [promoted build](http://download.java.net/glassfish/v3/promoted/) or [nightly build](http://download.java.net/glassfish/v3/nightly/) of the GlassFish v3 Preview application server.
2. If you haven't already done so, download [Maven 2](http://maven.apache.org/download.html).
3. Download the sample application package, [PostRedirectGet.zip](http://blogs.sun.com/enterprisetechtips/resource/PostRedirectGet.zip)
4. Extract the contents of the sample application package. You should see the folders prgViewParams, which contains the code for the PRG application that uses View Parameters.
5. Create the WAR file for the PRG application by changing to the prgViewParams directory and entering the following Maven command:
6. mvn install

|  |
| --- |
|  |

You should see the file prgViewParams.war in a newly-created target subdirectory under the prgViewParams directory.

1. Start the GlassFish v3 Preview application server by entering the following command:
2. <GFv3\_inst>/bin/asadmin start-domain

|  |
| --- |
|  |

where <GFv3\_inst> is where you installed the GlassFish v3 Preview application server.

1. Deploy the sample application. One way to do that is to copy the prgViewParams.war file to the <GFv3inst>/domains/domain1/autodeploy directory.
2. Execute the application by opening a browser and accessing the URL http://localhost:8080/prgViewParams. You should see the form shown in [Figure 1](https://blogs.oracle.com/enterprisetechtips/entry/post_redirect_get_and_jsf#fig1).

|  |
| --- |
| Registration Page  **Figure 1.***Registration Page* |

2. Enter information as appropriate into the form and click the Register button. You should see a page similar the one shown in [Figure 2](https://blogs.oracle.com/enterprisetechtips/entry/post_redirect_get_and_jsf#fig2).

|  |
| --- |
| Registering Through the Registration Page  **Figure 2.***Registering Through the Registration Page* |

2. Click the Confirm button. You should see a page similar the one shown in [Figure 3](https://blogs.oracle.com/enterprisetechtips/entry/post_redirect_get_and_jsf#fig3).

|  |
| --- |
| Confirmation Page  **Figure 3.***Confirmation Page* |



**Further Reading**

For more information, see the following resources:

* [JSR 314: JavaServer Faces 2.0](http://jcp.org/en/jsr/detail?id=314)
* [JavaServer Faces 2.0: The Complete Reference](http://bit.ly/5qrXJ)
* [Redirect After Post](http://www.theserverside.com/tt/articles/article.tss?l=RedirectAfterPost)
* [JSF 2.0 Bookmarkability/View Parameters](http://blogs.sun.com/rlubke/entry/jsf_2_0_bookmarability_view)

**About the Author**

Ed Burns is a senior staff engineer at Sun Microsystems. Ed has worked on a wide variety of client and server-side web technologies since 1994, including NCSA Mosaic, Mozilla, the Sun Java Plugin, Jakarta Tomcat and, most recently JavaServer Faces. Ed is currently the co-spec lead for JavaServer Faces. He is the coauthor of [JavaServer Faces: The Complete Reference](http://www.amazon.com/JavaServer-Faces-Complete-Reference/dp/0072262400) and the author of [Secrets of the Rockstar Programmers](http://www.amazon.com/Secrets-Rock-Star-Programmers-Riding/dp/0071490833). He is also the coauthor of the upcoming book [JavaServer Faces 2.0: The Complete Reference](http://bit.ly/5qrXJ). Read Ed Burns's [blog](http://www.java.net/blogs/edburns/).

Category: JavaServer Faces (JSF) technology

* Call [Flash#setKeepMessages()](http://docs.oracle.com/javaee/6/api/javax/faces/context/Flash.html#setKeepMessages%28boolean%29) with true before render response phase to instruct JSF to store the faces messages in the flash scope and add faces-redirect=true query string parameter to the outcome to perform a redirect.
* public String submit() {
* // ...
* FacesContext context = FacesContext.getCurrentInstance();
* context.addMessage(null, new FacesMessage(FacesMessage.SEVERITY\_INFO, "Success!", null));
* context.getExternalContext().getFlash().setKeepMessages(true);
* return "nextpage?faces-redirect-true";
* }
* This way there's no need for a phase listener which collects the faces messages from the faces context and stores them in the session before redirect and removes them from the session on the firstnext request and re-adds them to the faces context after redirect.
* The flash scope works roughly the same way. The messages are stored in the session by an unique identifier which is in turn been passed as a cookie in the response and those messages (and the cookie) are been removed from the session on the firstnext request which has passed the cookie back (which is, after all, a more robust implementation although the chance is very little that an enduser will send 2 HTTP requests on the same session at *exactly* the same moment — or it must be a robot).

### State Saving

The main problem for application developers is that the size of the saved state can be large. This makes client-side state saving impractical and leads to session state bloat. For component developers, the issue is that implementing [saveState](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/StateHolder.html#saveState%28javax.faces.context.FacesContext%29) and [restoreState](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/StateHolder.html#restoreState%28javax.faces.context.FacesContext,%20java.lang.Object%29) methods is tedious and error prone.

JSF 2.0 addresses these issues with the introduction of a new “partial” state saving mechanism. The key concept is that saving the entire state of the component tree is redundant, since the component tree can always be restored to its initial state by re-executing the view (ie. by re-executing Facelet handlers to re-create the component tree).

If we use the view definition to restore the component tree to its initial state, then the only state that needs to be saved is state that has been modified since the view was initially created. And since in most cases the number of components that are modified after component tree creation is small, the size of this delta state is typically much smaller than the full component tree state.

A requirement for the delta/partial state saving approach is that component implementations must know when their initial state has been fully configured. JSF 2 introduces the [PartialStateHolder](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/PartialStateHolder.html) contract to help with this requirement. The PartialStateHolder’s [markInitialState()](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/PartialStateHolder.html#markInitialState%28%29) method is called to notify the implementing component that its intial state has been established. Only modifications that occur after this notification need to be saved.

A second API has been introduced to help component implementations manage their state: [StateHelper](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/StateHelper.html). The StateHelper provides storage for component state (such as attributes, listeners, etc…) and relieves the component author from having to provide custom saveState and restoreState implementations.

As a result of these new APIs, JSF 2 state saving is both more efficient and easier to use!

If the PartialStateHolder/StateHelper APIs seem vaguely familiar to you, you just might be thinking of Apache Trinidad’s [FacesBean](http://myfaces.apache.org/trinidad/trinidad-api/apidocs/org/apache/myfaces/trinidad/bean/FacesBean.html).

### System Events

JSF 1.x includes two event delivery mechanisms. [FacesEvents](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/event/FacesEvent.html) are delivered to [FacesListeners](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/event/FacesListener.html) in response to user interaction, such as editing an input component, or activating a command component. [PhaseEvents](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/event/PhaseEvent.html) are delivered to [PhaseListeners](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/event/PhaseListener.html) at the beginning and end of each phase in the Faces lifecycle. These solutions do not include notification of other interesting events that occur during the processing of a Faces request.

JSF 2’s system events provide the ability to deliver [SystemEvents](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/event/SystemEvent.html) to [SystemEventListeners](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/event/SystemEventListener.html) at arbitrary points during the request processing lifecycle. System events fall into two categories: global system events and component system events.

Global system events are delivered in response to application-wide activities, for example [Application](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/application/Application.html) initialization/destruction. Listeners for these events are registered via a call to [Application.subscribeToEvent()](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/application/Application.html#subscribeToEvent%28java.lang.Class,%20java.lang.Class,%20javax.faces.event.SystemEventListener%29).

Component system events are specific to individual component instances. For example, a component system event is delivered each time a component has been added to the view, or rendered, or validated. Component system event listeners are typically registered via a call to [UIComponent.subscribeToEvent()](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/UIComponent.html#subscribeToEvent%28java.lang.Class,%20javax.faces.event.ComponentSystemEventListener%29). However, component system events also bubble up to the Application, where they are re-delivered to Application-level listeners. This allows global monitoring of component-related events.

Both global and component system events are fired by calling [Application.publishEvent()](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/application/Application.html#publishEvent%28javax.faces.context.FacesContext,%20java.lang.Class,%20java.lang.Class,%20java.lang.Object%29).

In order to simplify listener registration for component-specific events, JSF also provides a declarative listener registration solution via the [<f:event>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/event.html) tag. To register a listener declaratively, simply place the <f:event> tag inside of the target component. For example:

|  |  |
| --- | --- |
| 1  2  3 | <h:inputText>    <f:eventtype="preValidate"listener="#{bean.doSomePreValidation}"/>  </h:inputText> |

This registers the method located at #{bean.doSomePreValidation} as a pre-validation listener for the parent inputText component.

By providing notification of all sorts of activities that previously went undetected, system events open up many possibilities for page authors, component authors, framework authors and also for the JSF implementation itself!

### Navigation

#### Implicit Navigation

In JSF 1.x, even the most trivial navigation cases required an entry in faces-config.xml. For example, when navigating from page1 to page2 in response to a “success” outcome on a command component in page1 required the following XML boilerplate code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <navigation-rule>    <from-view-id>/page1.xhtml</from-view-id>    <navigation-case>      <from-outcome>success</from-outcome>      <to-view-id>/page2.xhtml</to-view-id>    </navigation-case>  </navigation-rule> |

Fun stuff, right?

JSF 2.0 introduces a simplification that reduces navigation-related grunge code: implicit navigation. If no matching navigation case is found after checking all available rules, the navigation handler checks to see whether the action outcome corresponds to a view id. If a view matching the action outcome is found, an implicit navigation to the matching view occurs.

The end result: smaller faces-config.xml files. Oh, and happier JSF users.

#### Conditional Navigation

The conditional navigation feature allows navigation cases to specify a pre-condition that must be met in order for the navigation case to be accepted. The pre-condition is specified as an EL expression using the new <if> configuration element:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <navigation-case>    <from-outcome>success</from-outcome>    <to-view-id>/page2.xhtml</to-view-id>      <!-- Only accept this case if the following condition is true -->    <if>#{foo.someCondition}</if>    </navigation-case> |

One of the motivations behind this feature is to allow pre-conditions to be implemented in a loosely coupled, navigation-agnostic way. Instead of requiring a managed bean or business layer object return a navigation-centric outcome value, we can now use EL to interact with arbitrary properties/methods on these objects, freeing these objects from awareness of the JSF navigation system.

#### Preemptive Navigation

The JSF 1.x navigation system is a black box. The only entry point, [NavigationHandler.handleNavigation()](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/application/NavigationHandler.html#handleNavigation%28javax.faces.context.FacesContext,%20java.lang.String,%20java.lang.String%29), simply evaluates the navigation rules and causes a navigation to occur, without giving any insight into how the navigation target is determined.

JSF 2.0 provides more transparent view of the navigation system. The new [ConfigurableNavigationHandler](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/application/ConfigurableNavigationHandler.html) API provides access to metadata describing the available navigation rules/cases. In particular, the [getNavigationCase](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/application/ConfigurableNavigationHandler.html#getNavigationCase%28javax.faces.context.FacesContext,%20java.lang.String,%20java.lang.String%29) method allows clients to ask the ConfigurableNavigationHandler to determine which navigation case matches a particular outcome and from action. With this new contract, it is possible to “tr” evaluate navigation rules and retrieve the resulting target view id and URL.

Why is this interesting? Well, before JSF 2.0, navigation rules were explicitly the domain of POST requests. Previously, the only time that navigation rules came into play was during the invoke application phase while handling a POST. By making the navigation rules available outside of invoke application, we open up the possibility of leveraging this information at other points in the lifecycle, for example, during render response.

As we’ll see in the next topic, this has some interesting consequences relating to better support for GET-based navigation and bookmarkability, an area that has received much interest in recent years.

#### Links

* [Issue 179](https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=179) Improve NavigationHandler API
* [Issue 419](https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=419) outcome->viewId (was: Make navigation rules more flexible)
* [Issue 454](https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=454) Decoupling navigation from the business layer
* [ConfigurableNavigationHandler](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/application/ConfigurableNavigationHandler.html) (JavaDoc)

### GET Support

It’s not so much that JSF 1.x does not provide any support for GET request handling… after all, there are some low-level primitives (like PhaseListeners) available. However, there is no denying that GET requests are a second-class citizen in earlier versions of the JSF specification. JSF 2.0 introduces several features that tackle this shortcoming.

#### View Parameters

The JSF 1.x specification goes to great lengths to provide a robust lifecycle for processing of values arriving from the client via Http requests. Values are pulled from the request, converted from strings to the proper target type, validated against application-specified constraints and finally, pushed into the model. Very powerful.

Of course, there is one caveat: all of this processing only applies to values arriving on POST requests. If the same value arrives via a GET request, well… tough luck! Or, well, maybe write a PhaseListener to manually process the request. Oh, and good luck reusing your converter/validator implementations to process those values!

JSF 2 brings some sanity to this situation with the introduction of “view parameters”, inspired by Seam’s [page parameters](http://docs.jboss.org/seam/1.1GA/reference/en/html/events.html#d0e3384). View parameters provide a simple, declarative way to map incoming request parameter values to any EL-reachable location. These mappings are specified using the new [<f:viewParam>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/viewParam.html) component, which lives in the (also new) [<f:metadata>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/metadata.html) section of the view, eg:

|  |  |
| --- | --- |
| 1  2  3 | <f:metadata>    <f:viewParamname="foo"value="#{bean.foo}"/>  </f:metadata> |

The above sample specifies that the value of the request parameter with the name “foo” will automatically be picked off of the request and pushed into the property at #{bean.foo}. So if we receive a GET request as follows:

page1.jspx?foo=bar

The value #{bean.foo} property will be set to “bar” when JSF starts processing the request. No more need to manually code this sort of mapping in a PhaseListener. Yay!

But wait, there is more… The <f:viewParam> component happens to be an [EditableValueHolder](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/EditableValueHolder.html), which means that view parameters can participate in conversion/validation. You can attach any converter/validator to an <f:viewParam> component , just as you would with, say, <f:inputText>. No more need to write custom conversion/validation logic specifically for GET processing.

#### PreRenderView Event

View parameters bring to GET requests much of the JSF request processing lifecycle that was formerly reserved for POSTs: request parameter decoding, type conversion, validation and model updates. But view parameters stop short of one last important step: invoking application-defined listeners. For this, [system events](https://andyschwartz.wordpress.com/2009/07/31/whats-new-in-jsf-2/#system-events) come to the rescue.

JSF 2 includes a [PreRenderViewEvent](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/event/PreRenderViewEvent.html) that is fired after view parameters have finished processing, but before the view is rendered. The pre-render view listener can be registered using the <f:event> tag, eg:

|  |  |
| --- | --- |
| 1  2  3  4 | <f:metadata>    <f:viewParamname="foo"value="#{bean.foo}"/>    <f:eventtype="preRenderView"listener="#{bean.doSomething}"/>  </f:metadata> |

This listener might be used to load data or set up context that is required prior to rendering. In addition, the listener may choose to programmatically navigate away from the requested view in the event that some pre-condition is not met (eg. the user does not have access to the requested page).

With the addition of view parameters and the pre-render view event, the GET request processing lifecycle is finally on par with the rich lifecycle provided for POSTs.

#### <h:link>/<h:button>

Now that we have the ability to respond to GET requests more effectively, what about the flip side? How do we issue these GET requests in the first place? Looking at the JSF 1.x components, <h:commandButton> and <h:commandLink> do not fit the bill as these components are form/POST-centric. <h:outputLink> works, but has the drawback that it requires manual construction of the destination URL.

JSF 2 includes two new components that simplify GET-centric navigation: [<h:link>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/link.html) and [<h:button>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/button.html). With these components, the page author no longer needs to hand code the destination URL. These components leverage the JSF navigation system to determine the appropriate destination. Instead of explicitly specifying the URL, users specify the desired logical outcome, eg:

|  |  |
| --- | --- |
| 1 | <h:linkoutcome="success"/> |

[Preemptive navigation](https://andyschwartz.wordpress.com/2009/07/31/whats-new-in-jsf-2/navigation-preemptive) is used to translate this logical outcome into a physical destination. At render time, the navigation system is consulted to map the outcome to a target view id, which is then transparently converted into the destination URL. This frees the page author from having to worry about manual URL construction.

Note that the URL construction process is not solely based on the target view id. The <h:link> and <h:button> components also integrate nicely with view parameters. By specifying the includeViewParams attribute, eg:

|  |  |
| --- | --- |
| 1 | <h:linkoutcome="success"includeViewParams="true"> |

The view parameters specified by the destination view are taken into account when building up the URL – ie. an inverse mapping of the view parameters is applied. Each view parameter value is extracted from the location specified by its EL expression, converted to a string, and added to the query string under the view parameter’s name.

This provides a simple yet powerful mechanism inserting required parameters into the query string. It also provides a nice technique for transferring request-scope data across requests.

In addition, page authors may manually insert query parameters using the <f:param> tag, eg:

|  |  |
| --- | --- |
| 1  2  3 | <h:linkoutcome="success">    <f:paramname="foo"value="bar"/>  </h:link> |

And of course, the key benefit of this solution: the resulting URLs are bookmarkable!

#### Links

* [Issue 487](https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=487) Add navigation controls
* [JSF 2.0 – Bookmarkability/View Parameters](http://blogs.sun.com/rlubke/entry/jsf_2_0_bookmarability_view)
* [<f:viewParam>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/viewParam.html) (Tag Doc)
* [<f:metadata>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/metadata.html) (Tag Doc)
* [<h:link>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/link.html) (TagDoc)
* [<h:button>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/button.html) (TagDoc)
* [PreRenderViewEvent](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/event/PreRenderViewEvent.html) (JavaDoc)

### Scopes

#### View Scope

View scope is one of two new scopes introduced in JSF 2 that fall into the category of “longer than request, shorter than session” scopes. As the name implies, state that is placed into view scope is preserved until the user finishes interaction with the current view. If you have ever had the joy of trying to figure out why your [command component isn’t firing](http://devgrok.blogspot.com/2009/06/commandbutton-not-calling-action-when.html) after binding the rendered attribute to a request-scoped managed bean, view scope may just be the answer to your problems.

View scope state can be accessed programmatically via [UIViewRoot.getViewMap()](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/UIViewRoot.html#getViewMap%28%29). EL access is available via the #{viewScope} implicit object.

#### Flash Scope

the new flash scope. Flash scope provides a short-lived conversation. State that is placed into flash scope is propagated across a single view transition, including surviving redirects, and is cleaned up before moving on to yet another view. If you have been resorting to session scope to store state that needs to survive a redirect, flash scope might just be for you.

Flash scope state can be accessed programmatically via the [ExternalContext.getFlash()](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/context/ExternalContext.html#getFlash%28%29) API. EL access is available via the #{flash} implicit object.

#### Custom Scopes

JSF 2 also provides a small enhancement to the managed bean system to allow faces-config.xml authors to place managed beans into custom scopes. The trick is that custom scopes are specified via an EL expression rather than a keyword, eg:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <managed-bean>    <managed-bean-name>foo</managed-bean-name>    <managed-bean-class>com.foo.Foo</managed-bean-class>    <managed-bean-scope>#{someCustomScope}</managed-bean-scope>  </managed-bean> |

The EL expression identifies the location of a map that holds the properties for the scope. When it is time to instantiate a custom-scoped managed bean, the managed bean system automatically pushes the new instance into the map under the specified name.

### Configuration

#### Managed Bean Annotations

JSF 2 provides a long awaited usability improvement with the introduction of annotation-based configuration. The goal of these annotations is to reduce the size and complexity of faces-config.xml files, which have a tendency to get quickly out of hand. The first set of annotations allows developers to configure managed beans. The old style XML configuration:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <managed-bean>    <managed-bean-name>foo</managed-bean-name>    <managed-bean-class>com.foo.Foo</managed-bean-class>    <managed-bean-scope>session</managed-bean>  </managed-bean> |

Can now be replaced with annotations on the bean class:

|  |  |
| --- | --- |
| 1  2  3  4 | @ManagedBean  @SessionScoped  publicclassFoo {  } |

The name for the managed bean is automatically derived from the name of the annotated class. In the above example, the presence of @ManagedBean annotation on the Foo class makes a managed bean with the name “foo” available. Alternatively, the @ManagedBean annotation also allows a name to be specified explicitly.

Annotations are also available for the other scopes, as well as for managed properties.

Note: An effort is underway to unify these bean/scope annotations across specifications (eg. JSF, [JCDI](http://jcp.org/en/jsr/detail?id=299)) for Java EE 6. In the meantime, the JSF 2 managed bean annotations are considered an optional part of the JSF 2 specification. Mojarra and MyFaces already provide implementations of these optional annotations.

#### Component Annotations

As part of the effort to prune down XML configuration bloat, JSF 2 also includes annotations targeted at authors of custom components (and associated objects). These annotations include:

* @FacesComponent
* @FacesRenderer
* @FacesConverter
* @FacesValidator
* @FacesBehavior

Of course, the good old faces-config.xml elements are still present for those who prefer to go that route.

#### Faces-config.xml Ordering

A long-standing problem with faces-config.xml loading is that the order in which these files are loaded is unspecified. For the most part (eg. for managed bean or navigation configuration) the order is not significant. However, there are certain cases, such as decorating Application-level objects (eg. ViewHandlers), where the order may matter.

During the JSF 1.2 timeframe, both MyFaces and the JSF RI adopted a convention whereby faces-config.xml files are loaded based on an order derived from the containing jar file’s name. However, this was just a temporary implementation-level solution that was put in place until the issue could be addressed by the specification.

JSF 2 solves this problem by allowing faces-config.xml files to provide ordering-related hints. Each faces-config.xml file may now declare a name (via a new <name> element) that can be referenced by other faces-config.xml files for ordering purposes. A new <ordering> element and sub-elements allow relative ordering requirements to be specified.

This addition to the spec provides a much cleaner/safer way for the various frameworks/component sets to play together in the JSF ecosystem. No more jar file name hacking!

#### Project Stage

Project stage is a new configuration option that allows the application developer/deployer to specify a hint to the JSF implementation regarding the role of the current deployment. Valid values include:

* Development
* Production
* SystemTest
* UnitTest

This hint allows the JSF implementation to optimize its behavior for the current stage. For example, the JSF implementation might provide more verbose development-time diagnostics than would be practical for a production environment.

For a more specific example, check out the Mojarra team’s recent use of project stage to provide [automatic compression](http://weblogs.java.net/blog/driscoll/archive/2009/07/automatic_compr.html) of JSF’s own JavaScript library.

The project stage can be set either as a context parameter (“javax.faces.PROJECT\_STAGE”) or via JNDI (“java:comp/env/jsf/ProjectStage”).

And, yes, my good friend and colleague [Matthias Wessendorf](http://matthiaswessendorf.wordpress.com/) would also like to see the project stage [exposed as a system property](https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=499). :-)

#### Links

* [Issue 121](https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=121) Require ordering of for loading META-INF/faces-config.xml files from component jar
* [Issue 287](https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=287) ConfigAnnotations – Annotations to obviate faces-config.xml
* [Faces-config.xml? … We don’t need no stinkin’ faces-config.xml!](http://blogs.sun.com/rlubke/entry/faces_config_xml_we_don)
* [JSF Configuration Resource Ordering](http://blogs.sun.com/rlubke/entry/jsf_configuration_resource_ordering)
* [JSF 2.0 New Feature Preview Series (Part 1): ProjectStage](http://blogs.sun.com/rlubke/entry/jsf_2_0_new_feature2)

### Behaviors

The original JSF 1.0 specification introduced the concept of “attached objects” – ie. objects that are not UIComponents, but are attached to UIComponents in order to influence some aspect of the component’s behavior. This technique is used for conversion and validation by value holding components, eg:

|  |  |
| --- | --- |
| 1  2  3 | <h:inputTextvalue="#{foo.value}">    <f:convertNumberintegerOnly="true"/>  </h:inputText> |

JSF 2 adds a new class of attached object: the [ClientBehavior](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/behavior/ClientBehavior.html). Whereas converters/validators focus on value processing, client behaviors assist in enhancing a component’s client-side functionality. Before client behaviors, page authors were forced to resort to hand-coding scripts and manually wiring these scripts up via event handling attributes, eg:

|  |  |
| --- | --- |
| 1 | <h:commandLinkonclick="return confirm('Really???')"/> |

Client behaviors allow such scripts to be packaged up into reusable bundles that can be attached to components declaratively, eg:

|  |  |
| --- | --- |
| 1  2  3 | <h:commandLink>    <foo:confirmmessage="Really???"/>  </h:commandLink> |

This declarative approach is made possible by the introduction of two new contracts (along with various supporting APIs). The ClientBehavior contract specifies how client behavior implementations provide access to scripts and more generally how these objects participate in the component lifecycle. The [ClientBehaviorHolder](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/behavior/ClientBehaviorHolder.html) contract specifies how client behaviors are attached to components, similar to how EditableValueHolder specifies how converters and validators are attached.

In addition to defining these new contracts, JSF 2 also includes one concrete ClientBehavior implementation: the [AjaxBehavior](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/behavior/AjaxBehavior.html). The declarative form of this client behavior should be familiar from our earlier discussion on Ajax:

|  |  |
| --- | --- |
| 1  2  3 | <h:commandLink>    <f:ajax/>  </h:commandLink> |

Although the JSF specification only defines a single concrete client behavior, the client behavior architecture is designed for extensibility. Expect to see plenty of third party client behavior implementations popping up!

#### Links

* [ClientBehavior](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/behavior/ClientBehavior.html) (JavaDoc)
* [ClientBehaviorHolder](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/behavior/ClientBehaviorHolder.html) (JavaDoc)
* [AjaxBehavior](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/behavior/AjaxBehavior.html) (JavaDoc)

### Validation

#### Bean Validation

The Bean Validation JSR ([JSR-303](http://jcp.org/en/jsr/detail?id=303)) defines a generic, tier-independent mechanism for specifying data validation constraints. The specification includes several standard constraint annotations (eg. @NotNull, @Size, @Min, @Max, etc…) and also allows custom constraints to be defined.

JSF 2 provides built-in integration with JSR-303 constraints. In environments where a bean validation implementation is present, JSF automatically validates constraints for beans that are referenced by UIInput values.

In addition, [<f:validateBean>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/validateBean.html) may be used to fine tune bean validation behavior. For example, the validationGroups attribute may be used to manually specify which validation groups should be taken into account when validating a particular component:

|  |  |
| --- | --- |
| 1  2  3 | <h:inputTextvalue="#{bean.foo}">    <f:validateBeanvalidationGroups="com.foo.validation.groups.Billable"/>  </h:inputText> |

When validation constraints fail, any associated error messages are automatically translated into FacesMessages by the JSF implementation, thus communicating the failures to the end user without placing any burden on the application developer.

#### Empty Field Validation

In previous JSF releases, Validators were not applied to EditableValueHolder components with null/empty submitted values. Unfortunately, this behavior limits the utility of constraints that actually check null/empty values, such as the JSR 303 @NotNull constraint. In order to support @NotNull and other similar constraints, JSF 2 changes the behavior of null/empty value validation. As of JSF 2, when a JSR-303 implementation is present, null/empty values are validated.

Since this may cause problems for legacy Validator implementations that do not expect to see null/empty values, the javax.faces.VALIDATE\_EMPTY\_FIELDS context parameter can be used to disable this behavior.

#### New Validators

In addition to <f:validateBean>, JSF 2 includes two other new validators:

1. [<f:validateRequired>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/validateRequired.html) provides required field validation.
2. [<f:validateRegexp>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/validateRegex.html) provides regular expression-based validation

#### Links

* [Issue 426](https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=426) Bean Validator support
* [Issue 480](https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=480) Optionally validate all fields including those that are submitted empty
* [JSR 303](http://jcp.org/en/jsr/detail?id=303) Bean Validation
* [<f:validateBean>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/validateBean.html) (Tag Doc)
* [<f:validateRequired>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/validateRequired.html) (TagDoc)
* [<f:validateRegexp>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/f/validateRegex.html) (TagDoc)

### Error Handling

JSF users who are new to Facelets will appreciate the more informative Facelets error page that provides information such as: line numbers! (And no, not just stack trace line numbers, but actual line numbers within the Facelets file.)

Of course, error pages are only useful in cases where the triggering exceptions are allowed to propagate up to the error handling layer and not pre-maturely consumed. The JSF 2 specification helps in this area with the introduction of the new [ExceptionHandler](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/context/ExceptionHandler.html) API. JSF implementations are now required to allow all “unexpected” exceptions to propagate out of the current lifecycle phase so that they can be handled globally by an ExceptionHandler instance.

This change should reduce/eliminate those mysterious, hard to debug “swallowed” exceptions, as the ExceptionHandler now acts as a central clearing house for exceptions. Another advantage of this centralization is that it allows users/frameworks to devise more sophisticated error handling strategies.

* [ExceptionHandler](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/context/ExceptionHandler.html) (JavaDoc)

### Resource Loading

If you spend time implementing custom JSF components of any complexity, eventually you are going to run into the question: what do I do with my component’s images (or JavaScript libraries, or style sheets)? Just about every JSF component framework has run into this problem and, in the absence of a standard solution, has solved this problem in its own proprietary way.

JSF 2 provides a standard solution to this problem with the introduction of the [ResourceHandler](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/application/ResourceHandler.html) API. The ResourceHandler is responsible for serving up resources (images, JavaScript files, style sheets, etc…) from well-known locations on the class path. Resource requests are routed through the FacesServlet, which passes these request onto the ResourceHandler for processing. This solution allows components and their resource dependencies to be bundled in the same jar file with no need for a bonus servlet, servlet filter or phase listener to serve up these artifacts.

JSF 2 also provides several new features in order to facilitate insertion of resource references. Authors of custom Java components can annotate their UIComponent subclasses with the @ResourceDependency annotation to identify any resources that need to be pulled in when the component is used. Composite component authors can use the new [<h:outputScript>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/outputScript.html) and [<h:outputStylesheet>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/outputStylesheet.html) to pull in scripts and style sheets. The #{resource} implicit object provides direct EL access to resource URLs.

The @ResourceDependency annotation and the <h:outputScript>/<h:outputStylesheet> components take advantage of another new feature: resource relocation. Resource relocation gives the component/page author control over where resource references are inserted into the rendered markup. For example, resource references may be inserted into the document head or at the end of the body. In addition to providing flexibility in where resources are inserted, a beneficial side effect of resource relocation is that duplicate resource references are automatically pruned out.

Note that in order to support resource relocation, JSF 2 introduces two other new components: [<h:head>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/head.html) and [<h:body>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/body.html) These components render the HTML head/body markup and also provide a target for relocatable resources.

#### Links

* [JSF 2.0 New Feature Preview Series (Part 2.1): Resources](http://blogs.sun.com/rlubke/entry/jsf_2_0_new_feature5)
* [JSF 2.0 New Feature Preview Series (Part 2.2): Resources](http://blogs.sun.com/rlubke/entry/jsf_2_0_new_feature)
* [JSF 2.0 New Feature Preview Series (Part 2.3): Resources](http://blogs.sun.com/rlubke/entry/jsf_2_0_new_feature3)
* [ResourceHandler](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/application/ResourceHandler.html) (JavaDoc)
* [<h:outputScript>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/outputScript.html) (TagDoc)
* [<h:outputStylesheet>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/outputStylesheet.html) (TagDoc)
* [<h:head>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/head.html) (TagDoc)
* [<h:body>](https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/h/body.html) (TagDoc)

### Tree Visiting

JSF 1.2 included the most useful addition of the [UIComponent.invokeOnComponent()](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/UIComponent.html#invokeOnComponent%28javax.faces.context.FacesContext,%20java.lang.String,%20javax.faces.component.ContextCallback%29) API. This provided the ability to invoke a callback on a single component instance in context. This functionality is particularly handy for any case where a single component (or subtree) needs to be targeted. (Think Ajax.)

While invokeOnComponent() is an excellent solution for the single component case, it is not particularly optimized for the multiple component case. If you need to target two components (eg. need to render two separate subtrees while processing an Ajax request), invokeOnComponent() will work, but requires two tree traversals in order to locate the two components. What happens if you need to render five separate subtrees? Or, even worse, what if you want to visit every component in context? Unfortunately invokeOnComponent() is not the answer.

JSF 2.0 solves the multi-component visiting case by introducing a sibling API to invokeOnComponent(): [UIComponent.visitTree()](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/UIComponent.html#visitTree%28javax.faces.component.visit.VisitContext,%20javax.faces.component.visit.VisitCallback%29). Both of these APIs traverse the component tree and invoke a callback on a subset of components. While invokeOnComponent() is designed to visit a single component, visitTree() may visit multiple components – and possibly all components – during its traversal. The set of components to visit is specified via a [VisitContext](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/visit/VisitContext.html) that is passed into the visitTree() call.

While tree visiting is currently used under the covers by JSF implementations for features such as Ajax and state saving, the hope is that JSF frameworks/users will find other use cases where this new API proves helpful. If you currently have any case where you are calling invokeOnComponent() repeatedly (or are calling findComponent() when you should be calling invokeOnComponent()), be sure to check out the new tree visitor API!

#### Links

* [Issue 180](https://javaserverfaces-spec-public.dev.java.net/issues/show_bug.cgi?id=180) Tree Visitor Pattern
* [New Feature for JSF 1.2](http://weblogs.java.net/blog/jhook/archive/2006/02/new_feature_for.html) (Jacob’s introduction to invokeOnComponent()
* [UIComponent.visitTree()](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/UIComponent.html#visitTree%28javax.faces.component.visit.VisitContext,%20javax.faces.component.visit.VisitCallback%29) (JavaDoc)
* [VisitContext](https://javaserverfaces.dev.java.net/nonav/docs/2.0/javadocs/javax/faces/component/visit/VisitContext.html) (JavaDoc)

### Conclusion

When I starting thinking about writing this article, I was hoping to provide a comprehensive overview of the entire JSF 2 feature set. As I got further into this undertaking, I realized that covering every single new feature/API in a single blog entry might be a bit, well… unwieldy. As such I decided to trim back my plans, but just slightly. There are a small number of minor API additions that I have not discussed here. Perhaps I’ll add a new section on these minor APIs at some point. For the moment, I am hoping that the information provided here is sufficiently complete to meet my original goal of providing a good introduction to the JSF 2 feature set.

If I have missed some feature/API that you think deserves discussion here, please leave a comment. Or if you know of other good sources of information, please leave a link. I am planning to update this blog from time to time with links to new blogs/articles as they become available, perhaps even to some that I write myself. :-)

[About these ads](http://wordpress.com/about-these-ads/)

### Related

[JSF 2 On The Road](https://andyschwartz.wordpress.com/2009/12/17/jsf-2-on-the-road/)With 2 comments

[What Do JSF Guys Do In Munich?](https://andyschwartz.wordpress.com/2009/11/14/what-do-jsf-guys-do-in-munich/)With 3 comments

[Back From AjaxWorld](https://andyschwartz.wordpress.com/2008/10/28/back-from-ajaxworld/)With 6 comments

This entry was posted on July 31, 2009 at 1:59 am and is filed under [Uncategorized](https://andyschwartz.wordpress.com/category/uncategorized/). You can follow any responses to this entry through the [RSS 2.0](https://andyschwartz.wordpress.com/2009/07/31/whats-new-in-jsf-2/feed/) feed. You can [leave a response](https://andyschwartz.wordpress.com/2009/07/31/whats-new-in-jsf-2/#respond), or [trackback](https://andyschwartz.wordpress.com/2009/07/31/whats-new-in-jsf-2/trackback/) from your own site.