Integrating Struts, Tiles, and JavaServer Faces

Bring the power, flexibility, and manageability of the three technologies together

Level: Advanced

Srikanth Shenoy (srikanth@srikanth.org), J2EE Consultant, Objectseek Inc.
Nithin Mallya (nithin@mallya.org), J2EE Consultant, Objectseek Inc.

September 23, 2003

Would you like the front-end power of JavaServer Faces (JSF), the content-formatting strengths of Tiles, and the flexibility of the Struts controller tier all wrapped up in your J2EE Web application? Enterprise Java experts Srikanth Shenoy and Nithin Mallya show you how to integrate the features of all three. This article demonstrates how to customize the classes in the Struts-Faces integration library to make them work with Tiles and JSF, explains the rationale behind doing this, and details how to use the new set of classes with a working example.

Contents:

A look at JSF
Why integrate the trinity?
Integrate Struts and JSF with Struts-Faces
Migrating Struts applications to JSF
Challenges
The changes so far
Resources
About the authors
Rate this article

Related content:

Struts, an open-source MVC implementation
Struts and Tiles aid component-based development
UI development with JavaServer Faces
Subscribe to the developerWorks newsletter
developerWorks Toolbox subscription

Also in the Java zone:

Tutorials
Tools and products
Code and components
Articles

By using Struts, Tiles, and JavaServer Faces (JSF) together, developers can ensure a robust, well-presented Web application that is easy to manage and reuse.

The Struts framework has been around for quite some time and has become the de facto standard that developers turn to when developing a J2EE Web application. The Tiles framework, which came soon after Struts, established its niche by offering developers the ability to assemble presentation pages using component parts. JSF, the newest kid on the Web-application-framework block, provides mechanisms for validating user input and handling user events; most importantly, it is a protocol-independent way of rendering user interface components. (For a quick look at these technologies, see the sidebar, "The major players.")

Although some of the functionalities in Struts and JSF overlap, they are complementary in other ways. The combination of these three technologies can provide an efficient way to develop a Web application, organize its presentation, and render custom user interface (UI) components independent of protocol.

To run the sample code from this article, you will need Struts 1.1, Tiles, JavaServer Faces Reference Implementation (JSF-RI) Early Access Release 4.0, and Struts-Faces 0.4. Struts and Tiles come bundled in the Struts 1.1 release from the Jakarta project. The Struts-Faces integration library can also be downloaded from the Jakarta project. The JSF-RI is part of the Web Services Developer Pack from Sun. (Links to these downloads and the sample code are available in Resources.)

And now, back to details of integrating the three technologies. First the bad news: as of the publication of this article, the three technologies do not interoperate out of the box. And the good news: in this article, we demonstrate how to integrate Struts, Tiles, and JSF. We assume that you already know Struts and Tiles. Some familiarity with JSF is helpful (see Resources for a link to a recent JSF

tutorial on developerWorks), but is not necessary to understand this article.

A look at JSF

JSF applications are normal J2EE Web applications using the JSF framework, a framework that provides a rich GUI component model that offers insight into how a real GUI framework should be. You might have heard people say that, although a certain technology is good, its look and feel still needs to mature. Well, the days of plain vanilla pages with HTML components are behind us and the days of having a superior GUI look and feel are here to stay if JSF has anything to do with it. How, you ask? Tree components, menu components, and graphs are some of the already existing UI components that JSF has to offer. Furthermore, JSF encourages the creation of custom components by providing an easy to use API.

Note: The UI components mentioned are part of the samples provided by Sun. As in any specification, the actual implementation is left to the various vendors.

In traditional Web applications that use the Model-View-Controller (MVC) pattern, the GUI components are represented by custom tags which handle both presentation and business logic. This leads to the problem of having to "code to a client device," which can involve duplication of code. Not so with JSF.

JSF architecturally separates the *presentation logic* (the "what") from the UI component's *business logic* (the "why" and "how"). By using JSF tags in your JSP pages, you can associate a renderer and a UI component together. A single UI component can be rendered in different ways by using different renderers. The UI component-specific code runs on the server and responds to events that are generated by user actions.

JSF-RI provides a render kit that comes with a custom tag library to render HTML from UI components. It also provides the ability to customize the look and feel of these components as desired. If specialized components are required, you can construct custom tags for a particular client device and associate it with a UI component and a custom renderer. For different devices, all you need to specify are different renderers.

JSF and the UI component

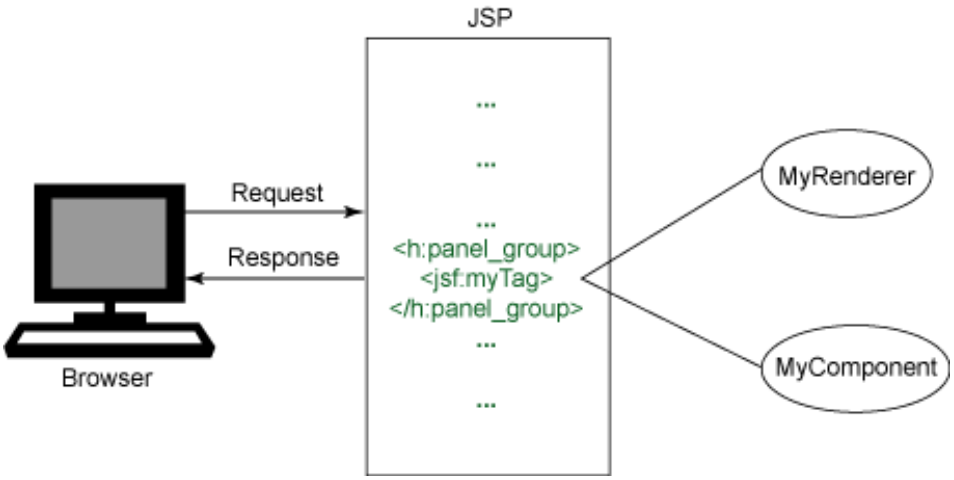
You may have created Java GUI applications using the Java AWT or Swing API, so you would be familiar with JSF's `UIComponent` (which is much like an AWT or a Swing component). It stores the tree of its child components (if they exist) and generates standard events for actions that occur on the client side, such as clicking a button to submit a form. These events are cached in the `FacesContext`. You can associate handlers for each of these events using custom tags. For example, you have a custom `ActionListener` to handle user clicks or form submissions.

The JSF `UIComponent`, `Renderer`, and the tag always go hand in hand. All JSF custom tags are created by subclassing `UIComponentTag`. The `doStart` and `doEnd` methods are already implemented in the `UIComponentTag` class. You only have to provide additional functionality in these tag classes.

Figure 1 illustrates the relationship between the custom tag, UI component, and the renderer. The client browser accesses a JSP page with JSF tags (`jsf:myTag`) for the UI component (`MyComponent`). The UI component runs on the server and is rendered back to the client as HTML using the appropriate renderer (`MyRenderer`). The JSP page expresses the user interface components with custom tags in JSF-RI rather than coding them in HTML.

For instance, Figure 1 shows the usage of the `h:panel:group` tag. This tag is used to group a set of components under one parent. When used in combination with other panel tags such as `panel_grid` and `panel_data`, it generates mark-up for columns in HTML tables at run time. The JSF-RI-provided *html_basic* tag library is used to represent HTML components such as text fields, buttons, and the like.

Figure 1. Rendering a JSF page



The JSF life cycle

The JSF life cycle consists of six phases; an incoming request might go through all or none of the phases depending on the type of request, validation and conversion errors that occur in the life cycle, and the type of response. *Faces requests* originating from JSP pages are handled by the JSF framework and a *faces* or *non-faces response* is returned.

A *faces request* occurs when a JSF form is submitted or when a user clicks a link that points to a page with the prefix */faces* in the URL. All faces requests are handled by a `FacesServlet` -- the controller servlet in JSF.

A request sent to a servlet or a JSP page with no JSF components is called a *non-faces request*. When the resulting page has JSF tags in it, it is called a *faces response*; with no JSF tags, it is a *non-faces response*.

There are six phases in the JSF life cycle:

- Reconstitute request tree
- Apply request values
- Process validations
- Update model values
- Invoke application
- Render response

According to the JSF specification, each phase represents a logical concept in the Request processing life cycle. However in the JSF-RI, these phases are represented by actual classes with corresponding names. The following section describes how each phase handles request processing and response generation. You will first see the phases involved in handling a faces request and then see the phases involved in handling a faces response.

Handling faces requests

To understand JSF request processing, look at FlightSearch.jsp, a simple JSF form in Listing 1. This is essentially how a JSF page looks. The JSF form has input text fields for *from* and *to cities*, *departure* and *return dates*, and buttons for submitting and resetting the form. (We'll examine what each tag in Listing 1 means shortly.) For now, assume that this form submission creates a faces request.

The request is received by the `FacesServlet` and goes through the various phases before a response is rendered back to the client. Figure 2 shows how a JSF request is processed. Let's see how this works.

1. Receive the request

The `FacesServlet` receives the request and gets an instance of the `FacesContext` from the `FacesContextFactory`.

2. Delegate life cycle processing

`FacesServlet` delegates the life cycle processing to the `Lifecycle` interface by invoking the `execute`

method on the `Lifecycle` implementation passing in the faces context.

3. Lifecycle executes each phase
The `Lifecycle` implementation executes each of the phases starting with the Reconstitute Component Tree phase.

4. Component tree created
In the Reconstitute Component Tree phase, a component tree is created with the components in `travelForm`. This tree has the `UIForm` as the root and the various text fields and buttons as its children.

The `fromCity` field has a validation rule that specifies that it cannot be empty, as shown by the `validate_required` tag. This tag links the `fromCity` text field with a JSF `Validator`.

JSF has several built-in validators. The corresponding `Validator` is initialized in this phase. This component tree is cached in the `FacesContext` and the context will be used in later phases to access the tree and invoke any event handlers. Also the `UIForm` state is saved automatically. So, when this page is refreshed, the form's original contents are displayed.

5. Extracting values from the tree
In the Apply Request Values phase, the JSF implementation traverses the component tree and extracts values from the request using the `decode` method and sets them locally for each of the components. If there are any errors during this process, they are queued on the `FacesContext` and will be displayed to the user in the Render Response phase.

Also, any events that were created as a result of user actions, such as clicking on the reset button, that are queued during this phase are broadcast to registered listeners. Clicking the reset button will set the values in the text fields back to their original values.

6. Validations are processed
In the Process Validations phase, any validations associated with each component are performed against the local values set in the Apply Request Values phase. This happens when the JSF implementation invokes the `validate` method on each registered validator.

If any of the validations fail, then the life cycle advances to the Render Response phase where the same page is rendered, but with the error messages. Here also, any events that are queued during this phase are broadcast to registered listeners.

The JSF implementation processes the validator on the source field. If the data is invalid, then control passes to the Render Response phase where FlightSearch.jsp is rendered again with validation errors displayed for the associated component. By declaring `output_errors` in the JSP page, all the errors in the page will be displayed at the bottom of the page.

7. Setting the model object values
In the Update Model Values phase, after all the validations are processed successfully, the JSF implementation sets the model object values with the valid ones by invoking the `updateModel` method on each component. If any errors occur while trying to convert the local data to the types specified by the model object properties, the life cycle advances to the Render Response phase where the errors are displayed. The values from the form field properties are populated into the model object's attribute values.

8. ActionListener can be invoked
You can associate an `ActionListener` with a user action such as clicking the submit button, as shown in Listing 1. In the Invoke Application phase, the `processAction` method is invoked on `FlightSearchActionListener`. Upon invocation, the `processAction` method would, in a real-world scenario, search the database for flights satisfying the criteria and retrieve the outcome from a component's action attribute.

In the example Web applications provided with this article, we have used static data to represent the list of flights. This method also sends the retrieved action attribute to the `NavigationHandler` implementation. The `NavigationHandler` looks up the faces-config.xml file -- the default application configuration file for JSF --

to determine the next page to be directed to based on this outcome.

9. Rendering the response
In the Render Response phase, the page obtained from the lookup in the faces configuration file, FlightList.jsp, is displayed if there are no errors in the faces context. If control came to this phase due to errors in any previous phases, then FlightSearch.jsp is redisplayed with the error messages.

Figure 2. Processing a JSF request
Click here to view the figure.

Listing 1. FlightSearch.jsp, a simple JSF form

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<f:use_faces>
 <h:form id="flightForm" formName="flightForm" >
   <h:input_text id="fromCity" valueRef="FlightSearchBean.fromCity">
        <f:validate_required/>
   <h:input_text/>

   <h:input_text id="toCity" valueRef="FlightSearchBean.toCity">
   <h:input_text id="departureDate"
     valueRef="FlightSearchBean.departureDate">
   <h:input_text id="arrivalDate"
     valueRef="FlightSearchBean.arrivalDate">

   <h:command_button id="submit" action="success"
     label="Submit" commandName="submit" >
       <f:action_listener
         type="foo.bar.FlightSearchActionListener"/>
   </h:command_button>
   <h:command_button id="reset" action="reset" label="Reset"
     commandName="reset" />

   <h:output_errors/>
 </h:form>
</f:use_faces>
```

Two tag libraries from JSF-RI are used in this code. The *html_basic* tag library defines tags for commonly used HTML components and the *jsf-core* tag library contains tags used to register listeners and validators. Other tags:

- The `f:use_faces` tag indicates to the JSF implementation that the tags following are faces tags.
- The `f:validate_required` tag indicates that the field to which it is attached (the `fromCity` field in `FlightSearchBean`) should have a value before the form can be submitted.
- The `h:form` and `h:input_text` tags represent an HTML form called `flightSearchForm` and the various text fields respectively.
- The `h:command_button` tag is used to represent Submit and Reset buttons.
- Finally, the `h:output_errors` tag is similar to the Struts `html:errors` tag and is used to display any errors that occur during validation of the form fields.

A JavaBean called FlightSearchBean represents the model that is updated during the Updated Model Values phase from the `UIComponent` data. Typically a JavaBean is declared in the JSP page with the `jsp:useBean` tag. You might notice that this has not been done in FlightSearch.jsp. This is because you can use a feature of JSF called *Managed Beans* whereby you declare all the JavaBeans components being used by the JSP pages in the faces configuration file. At startup, the servlet container initializes these JavaBeans components. The entry for FlightSearchBean in the faces-config.xml file is shown in Listing 2:

Listing 2. faces-config.xml entry for TravelInfoBean

```
<managed-bean>
  <managed-bean-name>FlightSearchBean</managed-bean-name>
  <managed-bean-class>
    foo.bar.FlightSearchBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Now let's look at the phases as they handle a response.

Rendering faces response
A faces response is generated by a Faces application when the resulting JSP page contains JSF tags. This response could be the result of a faces or non-faces request on a JSF application.

In our example, the rendering of the page in Listing 1 is a faces response. You might be familiar with the `doStartTag()` and `doEndTag()` methods on the `Tag` interface. In JSF and Struts-Faces, each tag extends from `UIComponentTag`. `UIComponentTag` implements the `doStartTag()` and `doEndTag()` methods.

It also has two abstract methods, `getComponentType()` and `getRendererType()`. By implementing these two methods in the concrete tag classes, you can specify the type of the component and the renderer, respectively.

Consider a simple JSF form with a text field. The following sequence of steps is executed when the JSF form is rendered.

1. Invoking the doStartTag() method
The Servlet container invokes the `doStartTag()` method on `FormTag`.

2. Getting the UIComponent
The `FormTag` gets its `UIComponent` from the `getComponentType()` method. `UIComponentTag` (parent of `FormTag`) uses `getComponentType()` to look up the class name for this component from the faces-config.xml file and creates an instance of the `UIComponent(FormComponent)`.

3. Getting the renderer
Next, `FormTag` gets its `Renderer` from the `getRendererType` method. As in the component type, the renderer name is looked up in the faces-config.xml file.

4. Encoding methods are invoked
After the `FormComponent` and `FormRenderer` have been created, the `encodeBegin()` method is invoked on the `FormComponent`. For each tag, the rendering begins with `encodeBegin()` and ends with `encodeEnd()`. The `encodeBegin()` methods are invoked in the order of nesting.

5. Ending tags and rendering HTML
The servlet container invokes the `doEndTag()` method on the tags. The `encodeEnd()` methods are invoked in the reverse order of nesting on each component. In the end, the Form and all nested components are rendered as HTML. At this point, the generation of HTML is complete and the HTML equivalent of the JSP is rendered.

Figure 3 shows the sequence of events that makes up the generation of a faces response.

Figure 3. Rendering a faces response
Click here to view the figure.

Why integrate the trinity?
As the JSP and the related specifications mature, new standards like JSF and the JSP Standard Tag Library (or JSTL, which uses simple tags to encapsulate the core functionality common to many JSP applications) are emerging. Following are some of the advantages to using the new technologies as an integrated whole:

- **Cleaner separation of behaviors and presentation.** With the separation of tag, renderer, and component, the roles of page authors and application developers in the development cycle become better defined.

- **Changing the presentation for a component does not have an avalanche effect.** Now you can easily just change the renderer. In the traditional MVC model, since this separation did not exist, any change in tags needed changes to the business logic as well. Not any more.

- **Renderer independence.** Or restated, protocol independence by reusing component logic for multiple presentation devices with multiple renderers. The ability to use different renderers eliminates the need to code the entire presentation tier for specific devices.

- **A standard for assembling and reusing custom components.** JSF thinks beyond "forms and fields" and provides a rich component model for rendering custom GUI components. Using JSF you can customize the way each component looks and behaves in a page. Developers also gain the ability to create their own GUI components (like menus and trees), which can easily be included in any JSP page with simple custom tags. Just like the Java front-end GUI components provided by AWT and Swing, we can have custom components for our Web pages that use their own event handlers and have customizable appearances. This is GUI nirvana for the Web tier!

Struts is a framework that already possesses a large customer base. Many IT departments have recognized the value of this MVC framework and have been using it for quite a while. JSF doesn't possess the equivalent of Struts's powerful controller architecture, as well as its standardized `ActionForm` and `Actions` (with their declarative capabilities). When you integrate Tiles into the mix, you give yourself the ability to reuse and change corporate layouts in a seamless manner.

The challenges of migrating JSF-enabled Struts applications are two-fold. First, Struts tags are not JSF-compliant. In other words, they do not extend the `UIComponentTag` as mandated by the JSF specification, therefore, JSF cannot interpret and associate `UIComponent` and `Renderers` with them.

Second, there is no link between the `FacesServlet` and Struts `RequestProcessor`. In a Struts application, the `RequestProcessor` manages the show with the callback methods into `ActionForm` and `Actions` classes. Getters and setters for `ActionForm` properties and `validate()` are the callback methods in the `ActionForm`. For `Action`, `execute()` is the callback method. Unless the `RequestProcessor` gets invoked, the callback methods in Struts `ActionForm` and `Actions` classes do not get a chance to invoke the business logic.

Integrate Struts and JSF with Struts-Faces
At this point you might be wondering if there is any software that can help integrate Struts with JSF, or whether you'll have to write the integration software yourself.

The good news is that the software already exists. *Struts-Faces* is an early access release of the Struts JSF integration library. This library was created by Craig McClanahan, the creator of Struts, and makes it easy to migrate your existing Struts applications to JSF (keeping the value of your existing Struts investment). Struts-Faces also strives for a clean integration with JSF so that JSF can be used on the front end while the back end will still have the familiar Struts components.

Figure 4 illustrates the relationships among Struts-Faces and JSF classes. The classes in blue belong to Struts-Faces.

Figure 4. Struts-Faces class diagram
Click here to view the figure.

The following are the major components of Struts-Faces:

- The `FacesRequestProcessor` class, which handles all faces requests. This class subclasses the regular Struts `RequestProcessor` and handles the faces requests. Non-faces requests are delegated to its parent, `RequestProcessor`.

- The `ActionListenerImpl` class, which handles `ActionEvents` such as submitting a form or clicking on a link. This class is used instead of the default `ActionListener` implementation provided by JSF-RI. Whenever an `ActionEvent` is generated in a faces request, the `processAction()` method on `ActionListenerImpl` is invoked and `ActionEvents` are forwarded to the `FacesRequestProcessor`. This is interesting because `RequestProcessor` is normally invoked only by the Struts `ActionServlet` to process HTTP requests.

- The `FormComponent` class, which extends from the JSF Form Component but is invoked within the Struts life cycle.

- A renderer and tag for the `FormComponent`.

- Tags and renderers for data that is rendered for output only, that is, where there is no need for a separate component. For instance, `ErrorsTag` and `ErrorsRenderer` are used to display the form errors in HTML.

- An implementation of the `ServletContextListener` called `LifeCycleListener`, which is used to register the appropriate `RequestProcessor` during initialization.

- The faces-config.xml file. This is already bundled in the struts-faces.jar file.

Listing 3 shows FlightSearch.jsp using the Struts-Faces tags. It is similar to the JSF example demonstrated in Listing 1. The differences are highlighted with boldface. In it, you will find that a new tag library, *tags-faces*, is added. This tag library definition declares the tags used by the Struts-Faces API.

Listing 3. FlightSearch.jsp using Struts-Faces tags

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-faces"
  prefix="s" %>

<f:use_faces>
    <s:form action="/listFlights">
     <h:input_text id="fromCity" valueRef="FlightSearchForm.fromCity"/>

     <h:input_text id="toCity" valueRef="FlightSearchForm.toCity"/>
     <h:input_text id="departureDate"
       valueRef="FlightSearchForm.departureDate">
     <h:input_text id="arrivalDate"
       valueRef="FlightSearchForm.arrivalDate">

     <h:command_button id="submit" action="success" label="Submit"
       commandName="submit" />
     <h:command_button id="reset" action="reset" label="Reset"
```

```
       commandName="reset" />

     <s:errors/>
    </s:form>
</f:use_faces>
```

The `s:form` tag is used to create the HTML form. The form action attribute is */listFlights* instead of the form name, *flightForm* as specified in Listing 1. In JSF, the form name is just a name assigned to the UIForm and nothing more.

`FlightSearchBean` is the model for the JSF Form and gets the values in Update Model Values phase. In Struts however, the form action points to the `ActionMapping` in the Struts Configuration File, struts-config.xml. To understand how this works you have to also take a look at the struts-config.xml file shown in Listing 4.

You will see that the `ActionMapping` for */listFlights* indicates that the `ActionForm` for this URI-path is `foo.bar.FlightSearchForm` and the `Action` class is `foo.bar.FlightSearchAction`. In other words, the `ActionForm` (`FlightSearchForm`) itself is the model for the HTML form in Struts-Faces and its action indirectly points to this model. (You can see this in Listing 3, where the text field tags point to the `FlightSearchForm`. In a normal Struts application this would have been `<html:text property="fromCity"/>`.)

Listing 4. Declaring the Action in struts-config.xml

```
<form-bean  name="FlightSearchForm"
              type="foo.bar.FlightSearchForm"/>

<!-- ========== Action Mapping Definition ========================== -->
<action-mappings>

<!-- List Flights action -->
 <action path="/listFlights"
    type="foo.bar.FlightSearchAction"
    name="FlightSearchForm"
    scope="request"
    input="/faces/FlightSearch.jsp">
    <forward name="success" path="/faces/FlightList.jsp"/>
 </action>

</action-mappings>
```

You will notice that the familiar *.do* is missing in the action attribute. This is because the Struts-Faces uses the form action itself as the form name (which should also match the `ActionForm` name in Struts configuration file).

Also notice that we have not used the JSF `validation` tag here. This is because in Struts, the validation happens in `validate()` method on the `ActionForm` class potentially by using the Commons-Validator. The `s:errors` tag is similar to the Struts errors tag and is used to display error messages that occur during validation.

Another thing to notice is that no `ActionListener` is explicitly associated with the Submit button. This is because the `ActionListener` is already provided in Struts-Faces and always forwards the faces requests with `ActionEvents` to the

Five steps to integrate Struts and Tiles

The following five steps will get Struts 1.1 and Tiles working together:

1. Create a JSP to represent your site layout. This is your master JSP with placeholders for header, body, and footers. Each of these are added to the main JSP page by using Tiles tags.

2. Create a Tiles definition file and

`FacesRequestProcessor`, from which the requests are dispatched to appropriate `Action` classes based on the struts-config.xml file.

Migrating Struts applications to JSF
In order to integrate the Struts Web application with JSF, follow these steps:

- Add the struts-faces.jar file along with the JSF-specific JARs (jsf-api.jar, jsf-ri.jar) into the *WEB-INF/lib* directory of the Web application.
- Add the JSTL-specific JARs (jstl.jar, standard.jar) into the *WEB-INF/lib* folder if you plan to use JSF and JSTL. This step is needed only if you are deploying to the regular Tomcat. JWSDP already provides these JARs.
- Modify the Web application deployment descriptor (*/WEB-INF/web.xml*) to have an entry for the Faces Servlet definition as shown in Listing 5.
- Modify the JSP pages to use the JSF and Struts-Faces tags instead of the Struts tags. Specifically replace the `html`, `base`, `form`, and `errors` tags with Struts-Faces equivalents. Replace the `text`, `textarea`, and `radio` tags with equivalent JSF tags. Struts-Faces does not have separate tags for these. Although not a requirement, you might also want to consider replacing the Struts Logic tags with equivalent JSTL tags.
- For each JSP that uses JSF tags, modify the struts-config.xml file to include the prefix */faces* in the *global-forwards* and the *local-forwards* in the Action Mappings pointing to that JSP.
- If the Web application uses any custom components that you've created, you will need to register them with the JSF implementation's default RenderKit. You can do this by creating a faces-config.xml file in the *WEB-INF* folder and adding entries for each component and renderer. However, remember that the faces-config.xml file is already bundled in the struts-faces.jar file. You have to extract it from the struts-faces.jar file, add your contents, and put it under *WEB-INF* folder.

define what JSP page has to be included in each of the placeholders for each aggregate page. Identify every aggregate page definition with a unique name.

3. Change the global and local forwards in the struts-config.xml file to use the unique names from the previous step instead of the aliases.

4. Use `TilesPlugIn` to load the Tiles definition file during startup. Add the `TilesPlugIn` entry into the struts-config.xml file.

5. Add the `TilesRequestProcessor` entry into the struts-config.xml file. This is the default request processor for Tiles-enabled Struts application.

Listing 5. Declaring the FacesServlet in web.xml

```
<!-- JavaServer Faces Servlet Configuration -->
<servlet>
<servlet-name>faces</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>

<!-- JavaServer Faces Servlet Mapping -->
<servlet-mapping>
  <servlet-name>faces</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Challenges to integrating Struts-Faces and Tiles
The Struts-Faces library provides an efficient bridge between Struts and JSF, making rich presentation layers a reality in J2EE Web applications. You can make the presentation layers even richer by adding Tiles to the combination, so that you not only get the benefit of the Struts and JSF combination, but you can also efficiently

reuse the various JSP pages because they will be made up of component parts or tiles that can be added or removed as required.

This article has already demonstrated the integration of Struts and JSF, so you would think that adding Tiles to the mix would be a breeze, right?

Unfortunately, JSF is still in the early stages and hasn't been posted in a final release. With this in mind, the Struts-Faces integration software is being developed iteratively to accommodate the various features of JSF and does not yet support Tiles.

Struts and Tiles can work together seamlessly, but you will encounter roadblocks on the integration journey. In the following subsections, you will find a summary of some commonly encountered issues you may face when using the Struts-Faces integration library in conjunction with Tiles. For each of these problems, we detail a solution by modifying the Struts-Faces classes. We will explain the solutions using the Flight Search example.

Listing 6 illustrates the layout for Flight Search page. Notice that we are calling it Flight Search page and not FlightSearch.jsp. This is because the FlightSearch JSP is the body of the aggregate page the user sees on the foobar travel Web site.

For now, we will retain the actual FlightSearch.jsp as is. We will change it as we go along. For your part, you will also have to create a Tiles definitions file with a definition for Flight Search page. Listing 7 (immediately following Listing 6) demonstrates an entry for Flight Search Page in the Tiles definition file. Notice the reuse of the master layout template with the `extends` attribute.

The individual potential challenges will follow Listings 6 and 7.

Listing 6. Tiles layout for the Flight Search example

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-faces"prefix="s" %>

<!-- Layout component parameters: header, menu, body, footer -->
<s:html>
<head>
  <title> <tiles:getAsString name="title"/></title>
  <s:base/>
</head>
<body>
  <TABLE border="0" width="100%" cellspacing="5">
    <tr>
     <td><tiles:insert attribute="header"/></td>
    </tr>

    <tr>
     <td><tiles:insert attribute="body"/></td>
    </tr>

    <tr><td><hr></td></tr>

    <tr>
     <td><tiles:insert attribute="footer" /></td>
    </tr>
  </TABLE>
</body>
</s:html>
```

Listing 7. Tiles definition for Flight Search Page.

```
<!-- Master Layout definition  -->
<definition name="foobar.master-layout"
  path="/faces/layout/MasterLayout.jsp">
      <put name="title"  value="Welcome to Foo Bar Travels" />
      <put name="header" value="/faces/common/header.jsp" />
      <put name="footer" value="/faces/common/footer.jsp" />
      <put name="body"   value="" />
</definition>

  <!-- Definition for Flight Search Page -->
<definition name="/foobar.flight-search"
  extends="foobar.master-layout">
      <put name="body"   value="/faces/FlightSearch.jsp" />
</definition>
```

Response has already been committed
This is the first problem you will see as soon as you try to access the Flight Search Form. Carefully look at the stack trace. You will see that the problem lies in the class `com.sun.faces.lifecycle.ViewHandlerImpl`. This is a JSF-RI class implementing the `ViewHandler` interface.

Figure 2 demonstrated the role played by `ViewHandler`. This is the class that forwards the request to the next page. When forwarding the request, it does not check the status of the response before forwarding it -- this happens only when you use Tiles, because Tiles internally includes the JSP pages in the response and JSF-RI commits the response after the first forward and then tries to forward again to the next Tiles include JSP.

To fix this problem, you will have to create a custom `ViewHandler` implementation that will check the status of the response to determine whether it has been committed. If the response has not been committed, then the request is forwarded to the next page; otherwise, the request is included and the appropriate JSP is displayed. We will create a class called `STFViewHandlerImpl` that implements the `ViewHandler` interface and implements the required method `renderView()`. Listing 8 shows the `renderView()` method in `STFViewHandlerImpl`:

Listing 8. renderView() method in STFViewHandlerImpl

```
RequestDispatcher rd = null;
Tree tree = context.getTree();
String requestURI = context.getTree().getTreeId();
rd = request.getRequestDispatcher(requestURI);

/** If the response is committed, include the resource **/
if( !response.isCommitted() ) {
   rd.forward(request, context.getServletResponse());
}
else {
   rd.include(request, context.getServletResponse());
}
```

Now that you have implemented your own `ViewHandler`, how do you notify the JSF-RI to use your `ViewHandler` instead of the default implementation? To answer this question, you have to understand the workings of `FacesServlet`.

During the Faces initialization process, the `FacesServlet` consults the `LifecycleFactory` implementation to return an implementation of `Lifecycle` class, as shown in Listing 9:

Listing 9. Faces initialization in FacesServlet

```
//Get the LifecycleFactory from the Factory Finder
LifecycleFactory factory = (LifecycleFactory)
  FactoryFinder.getFactory("javax.faces.lifecycle.LifecycleFactory");

//Get the context param from web.xml
String lifecycleID =
getServletContext().getInitParameter("javax.faces.lifecycle.LIFECYCLE_ID");

//Get the Lifecycle Implementation
Lifecycle lifecycle = factory.getLifecycle(lifeCycleID);
```

The `Lifecycle` implementation object holds the `ViewHandler` to be used during the Render Response phase. You can make your own `ViewHandler` implementation to be the default by calling the `setViewHandler` method on the `Lifecycle` implementation.

Now the question becomes how do you get the default `Lifecycle` implementation? The answer is that you don't need to do this. You just create a new implementation and register it with the `LifecycleFactory` with a unique ID, as shown in Listing 10:

Listing 10. Registering the custom ViewHandler and Lifecycle

```
//Get the LifecycleFactory from the Factory Finder
LifecycleFactory factory = (LifecycleFactory)
  FactoryFinder.getFactory("javax.faces.lifecycle.LifecycleFactory");

//Create a new instance of Lifecycle implementation -
//com.sun.faces.lifecycle.LifecycleImpl
//According to the documentation, factory.getLifecycle("STFLifecycle")
//should work, but JSF-RI has a defect.
//Hence this workaround of creating a RI class explicitly.
LifecycleImpl stfLifecycleImpl = new LifecycleImpl();

//Create a new instance of our STFViewHandler and set it on the Lifecycle
stfLifecycleImpl.setViewHandler(new STFViewHandlerImpl());

//Register the new lifecycle with the factory with a unique
//name "STFLifecycle"
factory.addLifecycle("STFLifecycle", stfLifecycleImpl);
```

You can see that the `lifecycleId` is hardcoded as `STFLifecycle`. Actually this is not the case. It becomes clear when you re-examine Listing 9. The `FacesServlet` gets the lifecycle ID from the context parameter declared in the web.xml file with the name `javax.faces.lifecycle.LIFECYCLE_ID` as follows:

```
    <context-param>
        <param-name>javax.faces.lifecycle.LIFECYCLE_ID</param-name>
        <param-value>STFLifecycle</param-value>
    </context-param>
```

Because the FacesServlet decides on the Lifecycle implementation class during its initialization, the code shown in Listing 10 should execute before the FacesServlet is initialized. You can do this by creating another servlet and initializing it before the FacesServlet.

But a smarter way to do this is by implementing a ServletContextListener interface. This class declares two methods, contextInitialized() and contextDestroyed(), which are called when the Web application is created and just before the Web application is destroyed, respectively. The code in Listing 10 is thus executed in the contextInitialized() method and the custom ViewHandler is already registered with the Lifecycle identified by the name STFLifecycle and is available to the FacesServlet. The ServletContextListener class itself is declared in the web.xml file as follows:

```
<listener>
  <listener-class>foo.bar.stf.application.STFContextListener
  </listener-class>
</listener>
```

This is not the only approach to registering a Lifecycle with custom ViewHandler. In fact the FactoryFinder implements its own discovery algorithm to discover the Factory objects, including the LifecycleFactory. These mechanisms include looking for the factory implementation class name in system properties, faces.properties file, or 1.3 Services discovery mechanism (META-INF/services/{factory-class-name}), in that order. However, the mechanism we've just discussed is the easiest and the most non-intrusive one.

404 Resource Not Found
After the committed response problem is fixed, click on any Tiles-specific link or enter a URL that would render a Faces response. In this case, you can enter the URL to display the FlightSearchForm.

Upon doing so, you get a *foobar.flight-search - 404 Resource Not Found* error. foobar.flight-search is the name of Tiles definition for Flight Search page. FacesRequestProcessor does not have the capability of processing Tiles requests (because it extends RequestProcessor instead of TilesRequestProcessor) and therefore, ends in an error.

To fix this problem, we will create a new request processor called STFRequestProcessor (stands for *Struts-Tiles-Faces Request Processor*). For now we will copy all the code from FacesRequestProcessor into the new class. The only difference is that STFRequestProcessor subclasses TilesRequestProcessor instead of subclassing the regular RequestProcessor. This new RequestProcessor can handle Tiles requests. Listing 11 details the STFRequestProcessor:

Listing 11. STFRequestProcessor.java

```
public class STFRequestProcessor extends TilesRequestProcessor
{

  protected void doForward(String uri, HttpServletRequest request,
    HttpServletResponse response)
  throws IOException,ServletException
  {
      //copy code from FacesRequestProcessor
  }

  protected void doInclude(String uri, HttpServletRequest request,
    HttpServletResponse response)
  throws IOException, ServletException
  {
      //copy code from FacesRequestProcessor
```

```
  }

  protected String processPath(HttpServletRequest request,
    HttpServletResponse response)
  throws IOException
  {
      //copy code from FacesRequestProcessor
  }

  protected String processPopulate(HttpServletRequest request,
    HttpServletResponse response)
  throws IOException
  {
      //copy code from FacesRequestProcessor
  }

  private void selectTree(FacesContext context, String uri)
  {
      //copy code from FacesRequestProcessor
  }

}
```

As you know, the `RequestProcessor` for Struts framework is specified in the struts-config.xml file. `STFRequestProcessor` becomes the processor when the following entry is added into the struts-config.xml file:

```
<controller processorClass="foobar.stf.application.STFRequestProcessor" />
```

Form submission displays the same form in return
Thanks to `STFRequestProcessor`, at this point you can navigate and see the Flight Search page. However, as soon as you submit the Flight Search form, you get the same form in return, but without the header and the footer! And there are no validation errors. In fact, there is no validation at all!

To get a hint of what is going on, go back to the Flight Search page and view the HTML source from the browser. You will see an entry like this:

```
<form name="FlightSearchForm" method="post"
  action="/flightapp/faces/FlightSearch.jsp">
```

Notice that the form action is pointing to the JSP page instead of a *.do*. Ah ha! There's the problem! This is not a new problem introduced when using Tiles with Struts-Faces; it is the default behavior of Struts-Faces to have the same JSP name as the form action. This behavior works without a hitch when you have a single JSP page (such as in the earlier Struts-Faces example). Listing 3 shows the original FlightSearch.jsp; let's go ahead and modify the action as follows:

```
<s:form action="/listFlights.do>
```

Of course, this modification alone does not solve the problem. With this change you will find that the `STFRequestProcessor` cannot find the `ActionForm`. Obviously more changes are required.

Before going ahead, though, look at Figure 5. It shows the relevant portions of the sequence of events in rendering

a faces response for a Struts-Faces form. This is same as in Figure 3 except for the highlighted method `createActionForm()` in `FormComponent`. The `FormComponent` class provided by the Struts-Faces API is a specialized subclass of `javax.faces.component.UIForm` and supports automatic creation of form Beans in request or session scope.

Figure 5. Rendering Struts-Faces response
Click here to view the figure.

As you can see, the `createActionForm()` method uses the action name itself to get the `ActionMapping` from the Struts configuration file. Because there is no `ActionMapping` for */listFlights.do*, Struts cannot find the `ActionForm`

The solution to this problem is to use `org.apache.struts.util.RequestUtils`. The static method `getActionMappingName()` in `RequestUtils` is intelligent enough to resolve the path (*/x/y/z*) or suffix (*.do*) mapping into appropriate `ActionMapping`.

Listing 12 shows the changes to the `createActionForm` method in boldface. Instead of doing these changes to the `FormComponent` in Struts-Faces, we create a new `STFFormComponent` by subclassing the `FormComponent` and overriding the `createActionForm()` method.

Listing 12. Modified createActionForm() method in FormComponent

```
// Look up the application module configuration information we need
ModuleConfig moduleConfig = lookupModuleConfig(context);

// Look up the ActionConfig we are processing
String action = getAction();
String mappingName = RequestUtils.getActionMappingName(action);
ActionConfig actionConfig = moduleConfig.findActionConfig(mappingName);
....
....
```

One more change is needed to the new `STFFormComponent`. Struts-Faces treats the action name itself as the form name. This needs to change because the action has the *.do* suffix in it while the form name does not have the *.do* suffix. So we add a new property called `action` to the `STFFormComponent` and override the `getAction()` and `setAction()` methods.

FormRenderer changes
You have to make a similar modification as the one shown in Listing 10 to the `encodeBegin` method of `FormRenderer` (the class that renders the Struts-Faces Form in HTML format).

Again, you do this by subclassing `FormRenderer`. In addition, you will also have to change the form action written out to the HTML. Listing 13 details these changes in boldface:

Listing 13. FormRenderer changes

```
protected String action(FacesContext context, UIComponent component) {

    String treeId = context.getTree().getTreeId();
    StringBuffer sb = new StringBuffer
      (context.getExternalContext().getRequestContextPath());
    sb.append("/faces");

    // sb.append(treeId); -- This is old code, replaced with
    // the two lines below.

    STFFormComponent fComponent = (STFFormComponent) component;
    sb.append(fComponent.getAction());

    return (context.getExternalContext().encodeURL(sb.toString()));
}
```

Changes to the FormTag

As you already know, when the component and renderers change, the tag has to change, too. In this case, create a new tag, `STFFormTag`, by subclassing from the `FormTag` in Struts-Faces. You don't have to change any of the functionality, just override the `getComponentType()` and `getRendererType()` methods. Listing 14 shows the overridden methods from `STFFormComponent`:

Listing 14. FormTag changes

```
public String getComponentType()
{
    return ("STFFormComponent");
}

public String getRendererType()
{
    return ("STFFormRenderer");
}
```

Modifying the faces-config.xml file

Custom components and renderers have to be declared in the faces-config.xml file so that JSF framework can instantiate and use them. We have created a new component, `STFFormComponent`, and a new renderer, `STFFormRenderer`, so far.

Now we will add the declarations to the faces-config.xml file as demonstrated in Listing 15. The *component-class* is the fully qualified class name for the component. The *component-type* refers to the name used in `STFFormTag` (Listing 12) to identify the component. Renderers are discovered and interpreted in similar manner. Note that the faces-config.xml file is present in the struts-faces.jar file. Remove the file from the struts-faces.jar file and put it under the *WEB-INF* folder of the Web application and modify it.

Listing 15. Declaring custom component and renderers in faces-config.xml

```
<faces-config>

  <!-- Custom Components -->
  <component>
    <component-type>STFFormComponent</component-type>
    <component-class>
      foobar.stf.component.STFFormComponent
    </component-class>
  </component>
  ..
  ..
  ..
  <!-- Custom Renderers -->
  <render-kit>

    <renderer>
      <renderer-type>STFFormRenderer</renderer-type>
      <renderer-class>
        foobar.stf.renderer.STFFormRenderer
      </renderer-class>
    </renderer>
    ..
    ..
    ..
  </render-kit>
</faces-config>
```

Modifying the struts-faces.tld file
You will not find the struts-faces.tld file in the sample Struts-Faces application; it is packaged along with the struts-faces.jar file. Open and examine it. It declares a class called `org.apache.struts.faces.taglib.LifecycleListener`, which implements `ServletContextListener` and initializes the `FacesRequestProcessor`.

Because you want to use the new `STFRequestProccessor`, you have to remove the file from the struts-faces.jar file, put it under the *WEB-INF* folder of the Web application, and delete the listener declaration. If you leave the tld file as is, then a `FacesRequestProcessor` will be instantiated in addition to the `STFRequestProcessor` when the Web application is initialized.

Modifying the base href tag
By now, you are past most of the hurdles in Struts, Tiles, JSF integration. You will even be able to navigate to the Flight Search page and enter your criteria and view the list of flights. Now try navigating back to the Flight Search Form from the Flight List page. You will get an HTTP 400 error. The reason for this error is the HTML `base href` tag. It is set to the Master Layout page.

```
<base href=
  "http://localhost:8080/stf-example/faces/layout/MasterLayout.jsp" />
           |_________|       |_____________________|
              Context              Servlet Path
```

All page navigations are being calculated relative to the layout page. It would be convenient if the `base href` tag included only up to the Web application context, like so:

```
<base href="http://localhost:8080/stf-example/" />
```

We can achieve this by customizing the Struts-Faces `BaseTag`. The changes in this class are pretty trivial. You just have to get rid of including the `HttpServletRequest.getServletPath()` in the base href.

Because these changes are display-related, a new renderer called `STFBaseRenderer` is created for it. The new tag is called `STFBaseTag` which declares `STFBaseRenderer` as its associated renderer. There is no need for a new Component.

With this information, the new `STFBaseTag` is created by subclassing the `BaseTag` and overriding the `getRendererType` method, as follows:

```
public String getRendererType()
{
    return ("STFBaseRenderer");
}
```

The changes so far

Congratulations! With these relatively minor modifications, you have successfully integrated Struts, Tiles, and JSF and saved any previous investment you might have made in these technologies. This article has demonstrated how to bring the front-end power of JSF, the content-formatting strengths of Tiles, and the flexibility of the Struts controller tier, together in one package to make crafting J2EE Web applications an easier task.

We've covered the customization of Struts classes to enable a tightly integrated working relationship with both JavaServer Faces and the Tiles framework, including such modifications and additions as:

- New `ViewHandler` to check for committed responses
- New `ServletContextListener` to create a new `Lifecycle` implementation and register the custom `ViewHandler`
- A new `RequestProcessor` to handle Tiles requests
- A modified web.xml file that declares the new `ServletContextListener` and the JSF Lifecycle ID
- New `FormTag`, `FormComponent` and `FormRenderer` classes
- New `BaseTag` and `BaseRenderer` classes
- A modified faces-config.xml file that declares the new component and renderer
- A modified struts-faces.tld file without the listener declaration

Hopefully, we've provided an overview of the component technologies used in this article, and more importantly, we've offered a cogent roadmap for you to combine Struts, Tiles, and JavaServer Faces into a powerful, flexible mechanism for building Web applications.

Resources

- Download the examples and code from this article and follow the instructions for build and deployment in README.txt.

- Ant is used to build the examples; you can download it from the Apache Ant Project Web site.
- For more about Struts and Tiles, including downloadable tutorials, documentation, binaries, and source code, try the Apache Jakarta Project Struts Web site.
- You can download the JSF Early Acess Release 4 (EA4) -- it comes with its own version of Tomcat -- as part of the Java Web Services Developer Pack Version 1.2.
- You can download version 0.3 or 0.4 of the Struts-Faces integration library from the Jakarta site.
- You can download JSF-RI in the Java Web Services Developer Pack 1.2.
- "Struts, an open-source MVC implementation" (*developerWorks*, February 2001) introduces Struts, a Model-View-Controller implementation that uses servlets and JavaServer Pages technology.
- "Struts and Tiles aid component-based development" (*developerWorks*, June 2002) explains why the Struts and Tiles combination is a terrific package of tools for creating Web applications and shows you how to get started using it, with a focus on changes since Struts 0.9.
- "Struttin' your stuff with WebSphere Studio Application Developer, Part 2: Tiles" (*developerWorks*, November 2002) is a tutorial that focuses on the use of the Tiles templating framework in conjunction with Struts using the WebSphere Studio Application Developer as the development environment.
- "Architect Struts applications for Web services" (*developerWorks*, April 2003) shows you how to build Web services applications based on the MVC design pattern using Struts.
- "A JSTL primer" (*developerWorks*, February-May 2003), a four-part series, offers all you ever wanted to know about JSTL, including how to use JSTL tags to avoid using scripting elements in your JSP pages, how to simplify software maintenance by removing source code from the presentation layer, and JSTL's simplified expression language, which allows dynamic attribute values to be specified for JSTL actions without having to use a full-blown programming language.
- Learn the basics for developing Web applications using JSF. In his tutorial, "UI development with JavaServer Faces" (*developerWorks*, September 2003), Jackwind Li Guojie explores the JSF life cycle, input validation, event handling, page navigation, and internationalization.
- Sun's JSF Web site is another good place to start to learn about JavaServer Faces technology.
- The ServerSide.com J2EE community is the ideal place to locate resources and participate in developer forums having to do with J2EE.
- The Java Community Process site is the place to go to get up to speed on the JavaServer Pages 1.2 specifications.
- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks* Java technology zone.

About the authors

Srikanth Shenoy specializes in the architecture, design, development, and deployment of large J2EE and EAI projects. He has helped clients in the manufacturing, logistics, and financial sectors to realize the Java's "write once, run anywhere" dream. He is a Sun Certified Enterprise Architect and co-author of the upcoming book *Practical Guide to J2EE Web Projects*. You can reach him at srikanth@srikanth.org.

Nithin Mallya specializes in providing enterprise solutions for financial clients. He has seven years of experience in architecting and developing server-side solutions, mostly for the Java platform. He is a Sun Certified Enterprise Architect and a Sun Certified Web Component Developer. He is a co-author of the upcoming book *Practical Guide to J2EE Web Projects*. You can reach him at nithin@mallya.org.

Integrating Struts, Tiles, and JavaServer Faces: The major players

A quick look at Struts, Tiles, and JavaServer Faces

Return to article.

Struts: A framework for developing Web applications using JSP technology (and part of the open source Jakarta project), Struts provides a flexible control layer based on standard technologies (such as Servlets, JavaBeans, XML, as well as various Jakarta Commons packages) and an application-architecture design based on the Model 2 approach, a variation of the Model-View-Controller (MVC) design. It provides its own Controller component and integrates with other technologies to deliver the Model and View components. Struts tags help to associate Bean properties with form fields, reducing the complexity of writing forms that remember the sum of user choices between requests.

Tiles: A framework that allows users to provide a consistent user interface, to display portlet-like rectangles of content within a larger page of content, and to download and process just one section of the image at a time, decreasing bandwidth needs. Through a central XML file that defines screens and a set of tags that can be embedded in JSP pages for the insertion of dynamic/static content, Tiles lets users build componentized views and assemble them as they choose.

JavaServer Faces: JSF technology makes it easier to build Web applications by letting users more easily assemble reusable UI components in a page, connect these components to an application data source, and wire client-generated events to server-side event handlers. JSF includes a set of APIs for representing UI components and managing their state, handling events and input validation, defining page navigation, and supporting internationalization and accessibility. It also includes a JSP custom tag library for expressing a JavaServer Faces interface within a JSP page. Learn more about JSF in the tutorial "UI development with JavaServer Faces" (see Resources).

Return to article.

Figure 2. Processing a JSF request

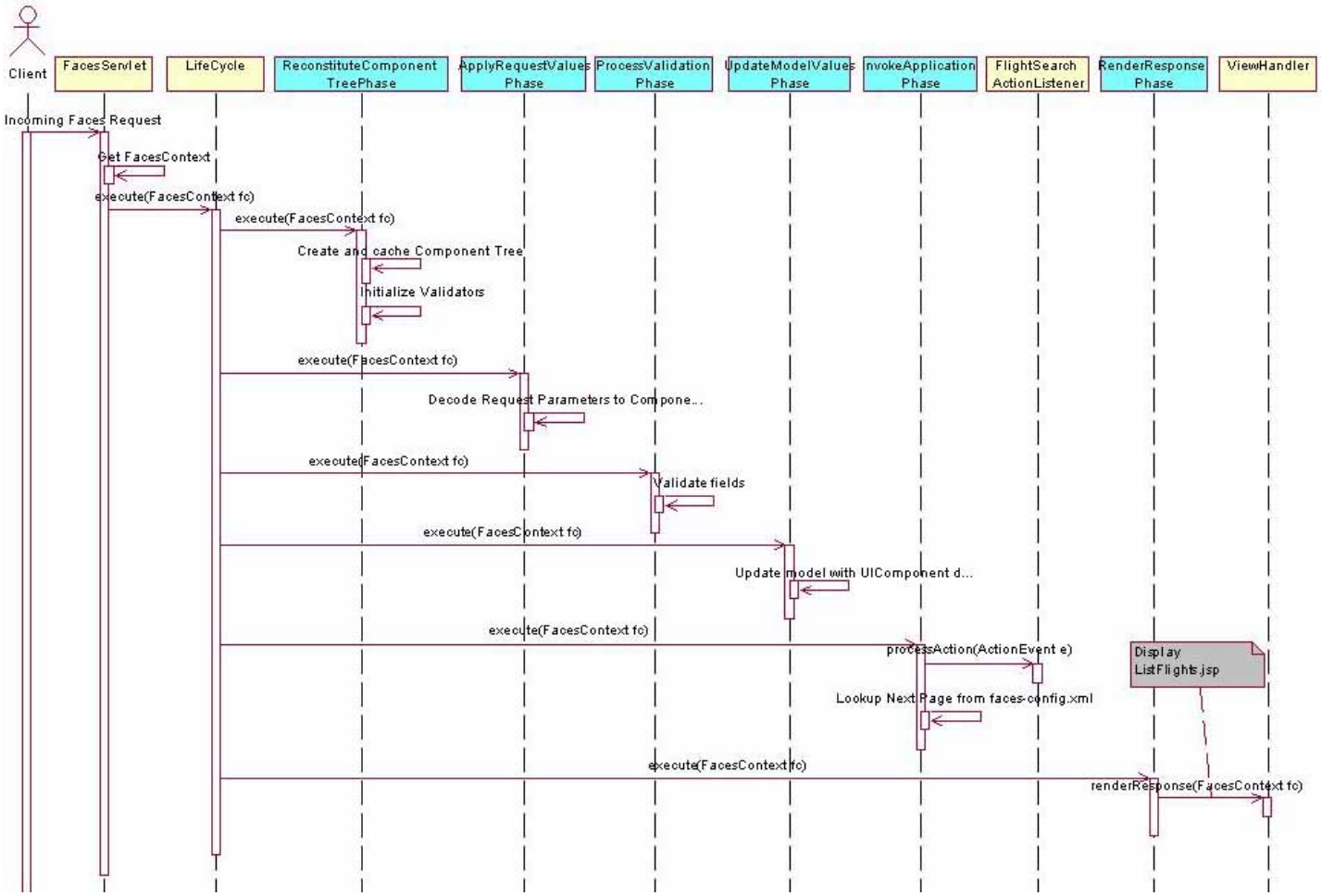
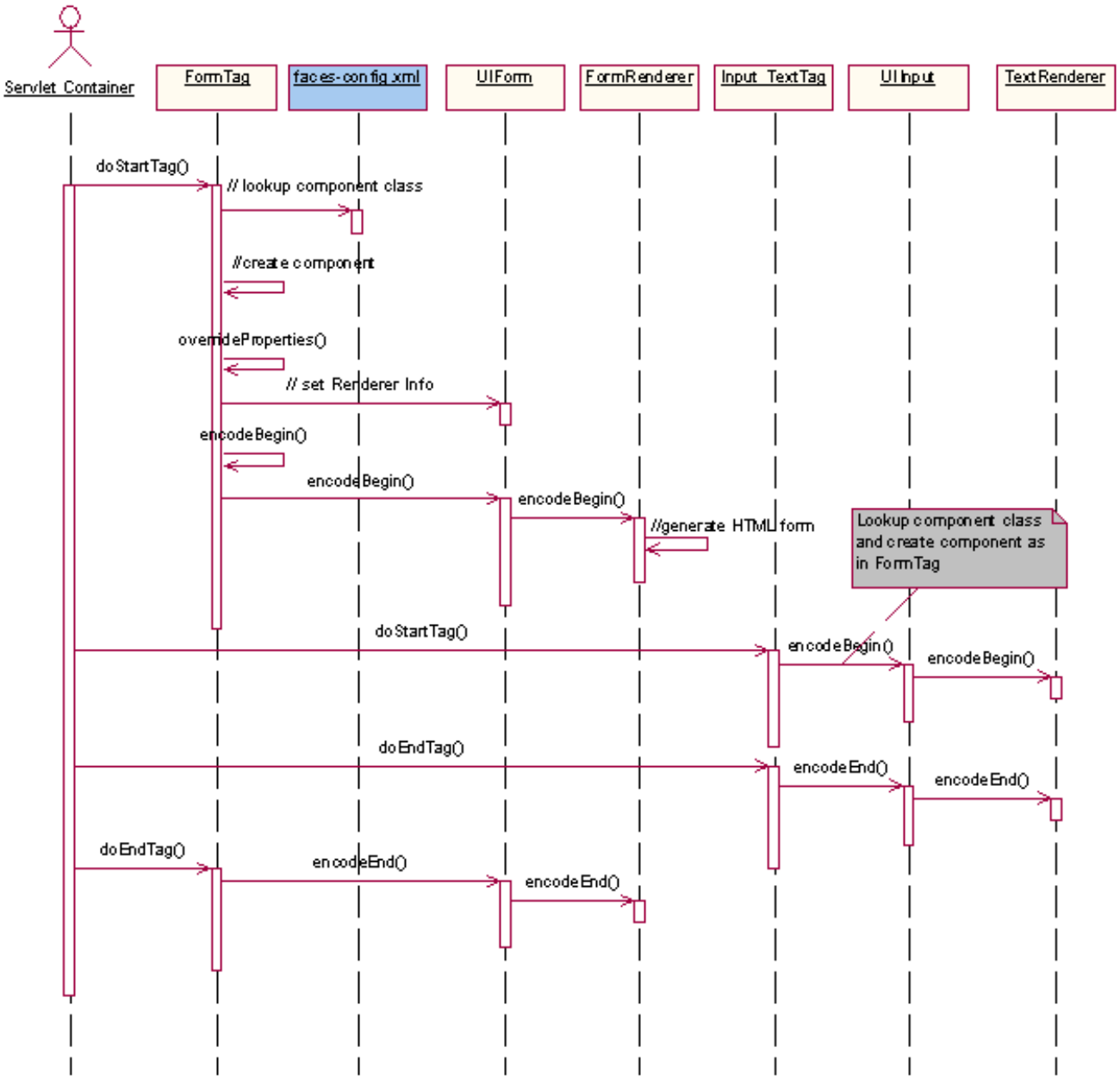


Figure 3. Rendering a faces response



Return to article.

Figure 4. Struts-Faces class diagram

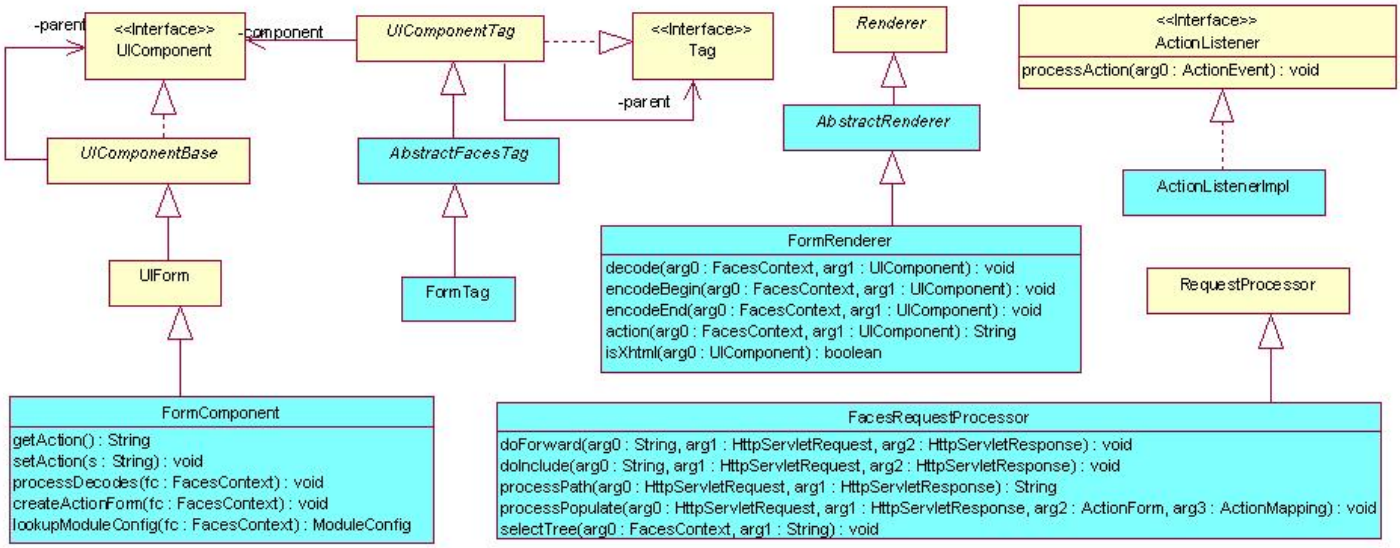
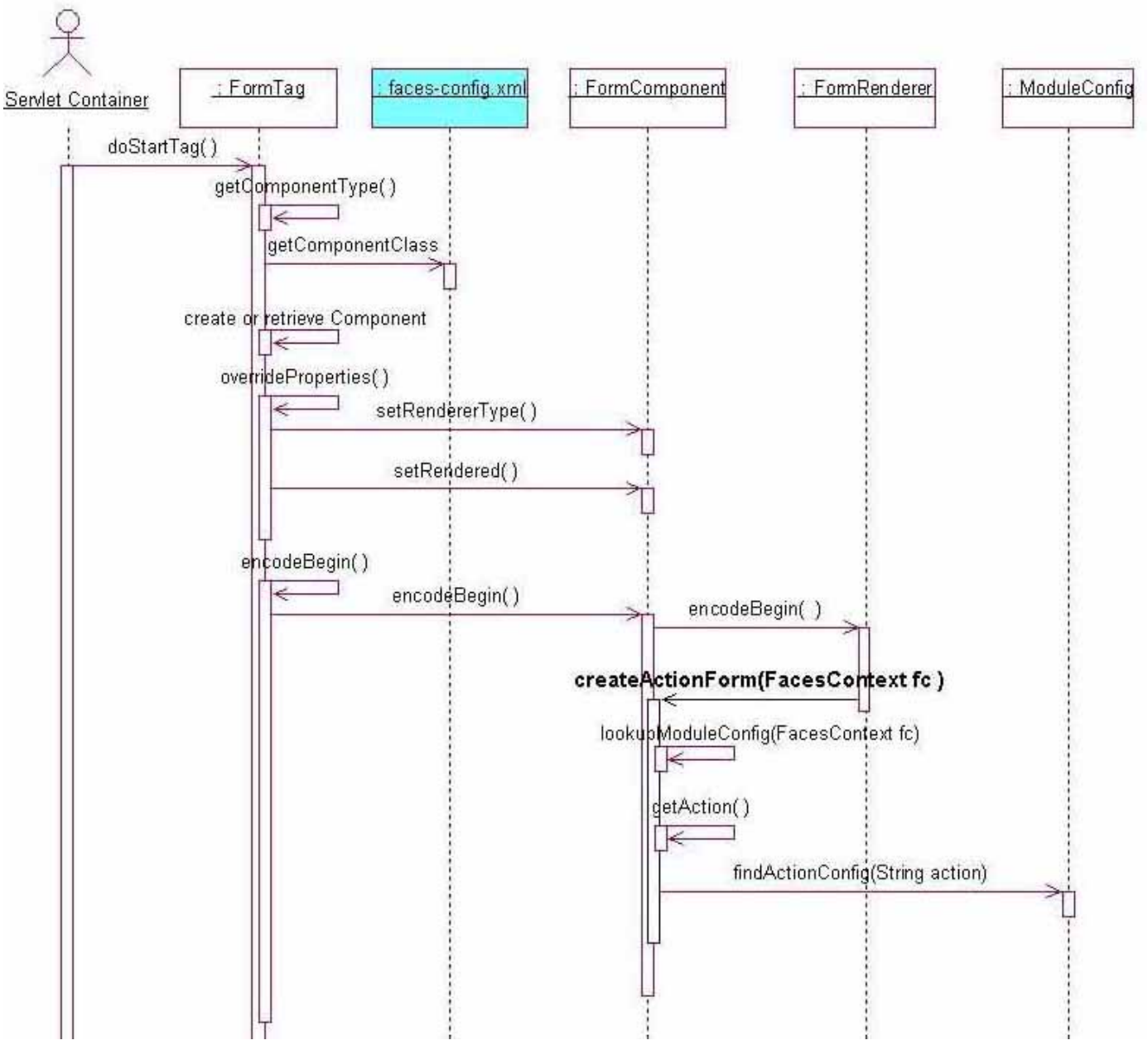


Figure 5. Rendering Struts-Faces response



Java technology

developerWorks
Java™ technology

Tools and products
Code and components
Projects
Education and events
Articles and columns
Forums
Standards
Special offers
News
Related links

More technologies
Grid computing
Linux
Open source projects
Security
Web architecture
Web services
Wireless technology
XML

Product domains
IBM developer solutions
DB2
eServer
Lotus
Rational
Tivoli
WebSphere
Toolbox subscription

IBM PartnerWorld
Benefits of joining
Technical resources and support
Products and technologies

JXTA 2
Building on his JXTA series *Making P2P interoperable*, Sing Li brings you up to date on the platform's major changes as it evolves and adapts to real-world topologies. (Articles)

- **Navigate the JNDI maze**: Write client code that successfully finds its way to an EJB component published in a JNDI namespace. (Articles)
- **Scheduling recurring tasks**: Learn how to build a simple, general scheduling framework for task execution conforming to an arbitrarily complex schedule. (Articles)
- **Java certification success, Part 1: SCJP**: This tutorial prepares you for the Sun Certified Java Programmer (SCJP) 1.4 exam, providing a detailed overview of the exam's main objectives and practice exercises to test your knowledge. (Education)
- **A practical introduction to TriActive JDO**: Learn how TJDO helps you to transparently persist data regardless of the underlying data store. (Articles)
- **Lock down J2ME applications with Kerberos**: Get in on a three-part series that shows you how to secure data with the industry standard, Kerberos. (Articles)

- **Specifications: Service Data Objects, WorkManager, and Timers**: IBM and BEA are collaborating on specifications for programming models and APIs for Java 2 Enterprise Edition (J2EE) application servers that provide programmers with simpler and more powerful ways of building portable server applications.
- **EclipseCon 2004 - The Premiere Conference on Open Tools Development and Integration**: See the breadth of Eclipse activity and interact with others in the community and the open source project. Mark your calendar for EclipseCon, February 2-5, 2004 in Anaheim, CA.
- **Write JMS programs using WebSphere**: Willy Farrell has updated his popular article on how to develop JMS programs with WebSphere MQ V5.3 and WebSphere Studio Application Developer V5. Be sure to see both Part 1 and Part 2.

Discussion forums

Join the discussion. Ask questions; get advice. Our Java programming experts keep these discussion forums on track. The forums now include several new functions -- enjoy!

- **XML and Java technology**: Want more on how these two technologies interact? XML/Java technology innovator Brett McLaughlin is here to help.
- **Java security**: Speak your mind on the Java security model. Security expert Paul Abbott is listening and can offer tips.
- **Java filter**: Not sure where to ask your question? Moderator Joe Sam Shirah can help or point you in the right direction.
- **Client-side Java programming**: developerWorks columnist John Zukowski answers your questions on topics like AWT, Swing, Java 2D, and others.
- **Server-side Java programming**: Programmer Govind Seshadri helps you resolve the tough challenges of server-side Java programming.
- **Multithreaded Java programming**: Do you need to understand the Java threading model? Brian Goetz can guide you through the maze.

Columns

Eye on performance by Jack Shirazi and Kirk Pepperdine
Get a better understanding of stress testing and the factors that go into choosing the right tool. (See previous columns.)

J2EE pathfinder by Kyle Gabhart
New! Kyle Gabhart provides a short introduction to the J2EE Web application security architecture. (See previous columns.)

Java theory and practice by Brian Goetz
New! Learn how the 1.4.1 JVM handles garbage collection, including some of the new garbage collection options for multiprocessor systems. (See previous columns.)

Magic with Merlin by John Zukowski
Merlin adds a subtle, but important change to JProgressBar, and John Zukowski shows you how to use it. (See previous columns.)

Most popular links

- IBM Developer Kits for AIX

Get started with Java technology
Tutorials
IBM developer kits
CDs and downloads
Submit content
IBM Redbooks
developerWorks journal
IBM developers' store

News

- Java tools organization pondered (InfoWorld)
- Rivals BEA and IBM deliver new Java specs (eWeek)
- IBM updates Rational Rapid Developer 2003 (Advisor.com)

alphaWorks code

Download and develop with:

- HeapRoots
- MBeanInspector for WebSphere Application Server
- JAR Class Finder
- Visual Application Builder

From IBM

- Running your Java application on AIX, Part 2: Gain a solid understanding of the JMM on AIX.
- Running your Java application on AIX, Part 1: Learn how the JIT compiler works and the implications of using JNI on AIX.
- A JSP ERP with DB2 Everyplace: A JSP Enterprise Resource Planning example solution implemented with IBM DB2 Everyplace.
- LOBs in DB2 UDB: A real-world example for using large object data types with DB2 Universal Database in your Java development.
- Develop embedded and mobile Java apps using UML: Achieve the highest levels of software quality with UML.
- Implementing an SQL EJB Wrapper as a Model Helper using an Access Bean: A sample implementation; also describes performance implications.

Powered by WebSphere

- Tutorial: UI development with JavaServer Faces
- Working with the Echo Web framework, Part 1
- Access USB devices from Java applications
- Simplify enterprise Java authentication with single sign-on

Webinar: What is your code REALLY doing? 2003 November 20	Webinar: What, When and How to Automate your Testing 2003 December 4