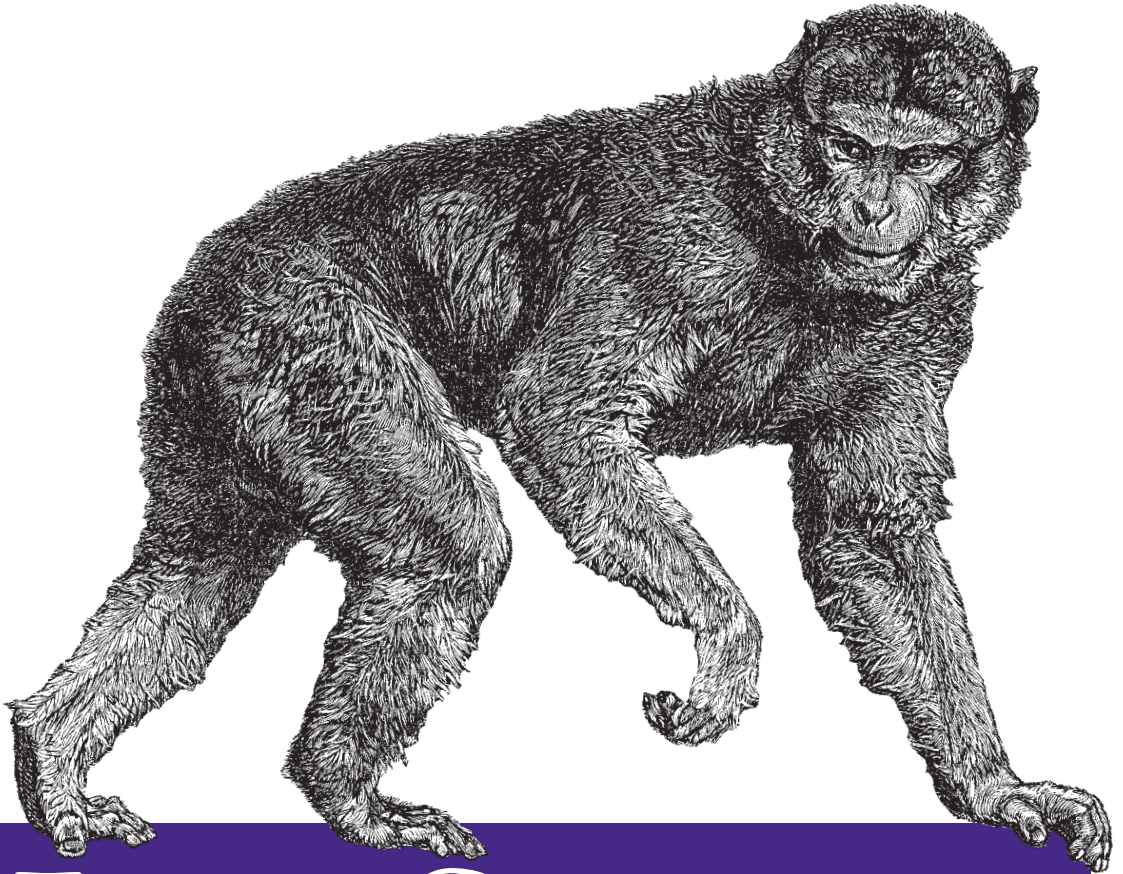


Building Web-based User Interfaces



JavaServer[™] Faces

O'REILLY[®]

Hans Bergsten

JSF Development Process Overview

Imagine building a flight reservation application with a web interface. The user first enters the departure and destination airports and dates, and preferences such as ticket type, airlines, and number of stops. This information is then validated and the user is presented with matching flight choices. When the user picks his preferred flights, the application ensures that the seats are available and marks them as reserved, calculates the cost, verifies credit card information, and finalizes the purchase.

People who are not computer gurus use applications like this, so the user interface must be intuitive and easy to use, error messages must be understandable, and the underlying problems must be simple to correct. For instance, the interface may let the user pick the destination airport by first asking for a country, state, or city name, and then present a selection of airports matching the criteria, provide calendars for choosing the dates, and display tables with flights to choose from. And the interface must be easy to enhance with accumulated user feedback and usage log analysis. The backend code requirements are also complex: accessing real-time flight schedules and reservation information, interfacing with credit card companies, and providing secure tracking of all transactions.

Clearly, this is not an application that can be slapped together without careful design. Applying the Model-View-Controller (MVC) principles briefly introduced in Chapter 1, we first break the application into classes that represent the business data and logic (the Model, including things like Customer, Airport, Flight, Seat, and so on), the user interface (the View, including things like Departure Input Field, Airport List, and so on), and the code that ties it all together (the Controller).

The View can be implemented in many different ways, using both client-side and server-side technologies. For instance, any of the traditional Java server-side technologies like JSP, Velocity, or plain servlets can render an HTML page with input fields, selection lists, and tables representing calendars. For complex user interfaces like this, however, the traditional technologies tend to result in pages with so much code that it becomes hard to make purely visual changes, such as changing the layout of a table.

With a JSF-based implementation, separate objects represent each user interface element, each keeping track of its UI state (e.g., brief or detailed display, number of rows to display, and the current start index) and its application data (e.g., the selected flight). The user interface objects also know how to render themselves in a way that can be customized based on developer or user settings, or even on the device type making the request. User actions, such as switching from brief to detailed display by clicking a button, are represented as events that can be handled either by the user interface object itself or an external event listener provided by the application developer. This type of event-driven component model has made it easier to develop very complex user interfaces for standalone applications over the years and, with a few twists, it simplifies web application development as well.

Developing an Application with a JSF-Based User Interface

To get an idea of how JSF simplifies development and maintenance of complex web-based user interfaces, let's implement a simple newsletter subscription application. The application contains a form where the user enters an email address and selects the newsletters of interest from a list, plus a button for submitting the form. Figure 2-1 shows this user interface.



Figure 2-1. Newsletter subscription form

When the user submits the form, the email address and subscription list is saved in a database. Other parts of the application use this information to send the newsletters, but we'll focus on this single page here.

JSF-based application development involves a number of different activities. It helps to define the different roles that the developers play, and then discuss the application aspects each role is responsible for. One person can, of course, take on more than one role. First, you need an implementation of the JSF framework itself. This is the responsibility of the *JSF implementor*, a role that's usually performed by web container vendors. Another role, the *tool provider*, is responsible for developing tools to support JSF application development. Some web container vendors take on this role

as well, but vendors specializing in development tools, such as Macromedia, are also likely candidates. Remember that JSF is a specification, not a product, so you have a choice of many competing implementations.

Most development projects use an existing JSF framework and tools, so let's focus on the remaining roles: the *application developer*, the *component writer*, and the *page author*. Implementing the newsletter subscription example shows you what parts each role is responsible for and gives you a glimpse of how JSF works.

Developing the Application Backend

The application developer is primarily responsible for developing the backend part of the application; in other words, the classes that handle business logic and data.

For the newsletter subscription form, the application developer may create a class called `Subscriber` to hold the subscriber information:

```
package com.mycompany.newsservice.models;

public class Subscriber {
    private String emailAddr;
    private String[] subscriptionIds;

    public String getEmailAddr() {
        return emailAddr;
    }

    public void setEmailAddr(String emailAddr) {
        this.emailAddr = emailAddr;
    }

    public String[] getSubscriptionIds() {
        return subscriptionIds;
    }

    public void setSubscriptionIds(String[] subscriptionIds) {
        this.subscriptionIds = subscriptionIds;
    }
}
```

The `Subscriber` class adheres to the JavaBeans method naming conventions—“properties” are defined by methods for getting their values, named `get` plus the name of the property, and methods for setting their values, named `set` plus the property name. As you'll see shortly, this makes it easy to use properties of this class as JSF UI component models.

When a subscription is registered or updated, the information must be saved somewhere, likely in a database. The application developer may decide to develop a separate class responsible for this task or put this logic in the `Subscriber` class. To keep the example simple, we'll add a method that represents this behavior to the

Subscriber class. Also, instead of saving the information in a database, we just write it to System.out:

```
public void save() {
    StringBuffer subscriptions = new StringBuffer();
    if (subscriptionIds != null) {
        for (int i = 0; i < subscriptionIds.length; i++) {
            subscriptions.append(subscriptionIds[i]).append(" ");
        }
    }
    System.out.println("Subscriber Email Address: " + emailAddress +
        "\nSubscriptions: " + subscriptions);
}
```

This is all the backend code we need for this simple application. Note, however, that none of these application classes refer to JSF classes; the same classes could be used with any type of user interface.

Developing Components and Integration Code

The component writer develops all application-specific Java code needed for the user interface, ranging from simple classes that tie the user interface to the application backend code developed by the application developer to custom user interface components when the ones provided by JSF aren't sufficient.

Figure 2-2 shows the main classes and interfaces used by the newsletter application.

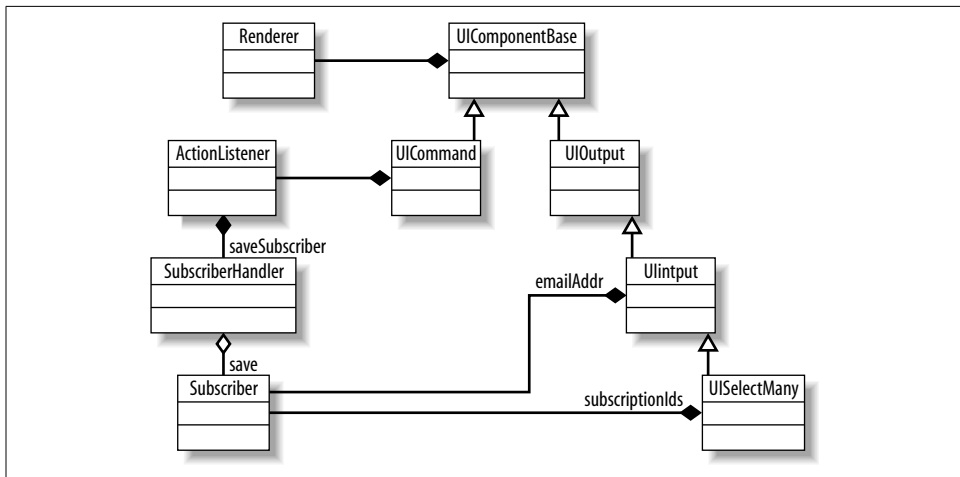


Figure 2-2. JSF component-related classes and the Subscriber application class

You probably recognize the Subscriber class from the previous section. The component writer develops the SubscriberHandler class, as shown in this section. All the other classes in Figure 2-2 are provided by the JSF implementation.

The `UIComponentBase` class is the base class for all JSF UI components. Subclasses represent specific interface elements, such as text fields, links and buttons, labels, menus and selection lists. JSF includes a set of component classes that can be used right out of the box, such as the ones shown in Figure 2-2, but a component writer can also implement custom components if needed. The `UIInput` class represents an input field and lets you bind the component to an application model through a *value binding*. When a component is rendered, it pulls its value from the application model object based on this value binding. Similarly, when an input component processes a request, it updates the model it's bound to with the value received with the request. In this example, value bindings for the `UIInput` and the `UISelectMany` (the checkbox group) components bind the components to the corresponding properties in the `Subscriber` application class.

The components fire events in response to user actions (such as clicking a button) and event listeners attached to the components handle the events (for example, by updating a database). Instead of implementing and registering listeners for each component, most JSF applications take advantage of shortcuts in the form of *method bindings*. A method binding is similar to a value binding, but it binds a component to an application method instead of an application property value. For instance, the `UICommand` component has a property that takes a method binding value. When the component fires an `ActionEvent`, a default `ActionListener` provided by JSF and automatically attached to the component invokes the method that the method binding points to. All the component writer needs to do is implement the method.

For the subscription example application, let's say that the component writer implements the action event processing method in a new class called `SubscriberHandler`:

```
package com.mycompany.newsservice.handlers;

import com.mycompany.newsservice.models.Subscriber;

public class SubscriberHandler {
    private Subscriber subscriber;

    public void setSubscriber(Subscriber subscriber) {
        this.subscriber = subscriber;
    }

    public String saveSubscriber() {
        subscriber.save();
        return "success";
    }
}
```

The `SubscriberHandler` class has two methods: a setter method for associating it with an instance of the `Subscriber` class and a method for handling the Save button `ActionEvent`. The `saveSubscriber()` method simply calls the `save()` method on the `Subscriber` instance and returns success. In a real application, it would return one or

more other values if things didn't go as planned, e.g., `systemFailure` if the database wasn't available.

It may seem like overkill to create a new class with the only purpose of calling a method in the application backend class, and in this simple example, it probably is. As you'll see when we start implementing a more complex application, there's a lot more that typically goes into this type of class. Its main purpose is to bridge the gap between the pure application objects and JSF components. For instance, it would be a bit odd for the `save()` method in `Subscriber` to return a `String`, but it serves a purpose for a JSF action event processing method to do so, because JSF can use the returned value to decide what to do next. But let's save that for later and move on to the next component writer task.

JSF creates and configures instances of application classes, such as the `Subscriber` and `SubscriberHandler` classes, based on information specified in a configuration file named `faces-config.xml`. It makes the instances available through variables that the page author uses in value and method binding expressions, as you'll see in the next section. Creating the configuration file is a task that also falls on the component writer's shoulders because it's part of tying the user interface and the backend together. Here's a snippet of the configuration file with the declarations for the example application classes:

```
<faces-config>
...
<managed-bean>
  <managed-bean-name>subscr</managed-bean-name>
  <managed-bean-class>
    com.mycompany.newsservice.models.Subscriber
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
<managed-bean>
  <managed-bean-name>subscrHandler</managed-bean-name>
  <managed-bean-class>
    com.mycompany.newsservice.handlers.SubscriberHandler
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>subscriber</property-name>
    <value>#{subscr}</value>
  </managed-property>
</managed-bean>
...
</faces-config>
```

As is customary these days, the configuration file is an XML document. The application objects are declared by `<managed-bean>` elements. The first `<managed-bean>` element declares that an instance of the `Subscriber` class should be made available as a session scope variable named `subscr`. I'll get back to the scope in Chapter 4, but declaring the

scope as session (as in this example) means that a unique instance is created for each user and remains available as long as the user actively uses the application.

The second `<managed-bean>` element contains a similar declaration of a variable named `subscrHandler` as an instance of the `SubscriberHandler`, but it also contains a `<managed-property>` element, saying that its `subscriber` property must be initialized with the value of the `subscr` variable. This is how the `SubscriberHandler` instance is linked to the `Subscriber` instance.

Developing the User Interface Pages

With the Java classes defined and implemented in some form (maybe just prototypes initially), the page author can get to work.

The page author is the person who is responsible for developing the pages that make up the application's user interface, typically as templates that interleave static content (text, graphics, tables for layout, and so on) with dynamically generated content. A page is represented by a set of UI component instances bound to the application's data and methods. The static content and the dynamic content generated by the components is combined and sent to the browser. When the user clicks a link or a button in the page, the request is processed by the methods bound to the UI components. Depending on the outcome, the same page may be rendered again or the application may select a new page to send back to the user.

As I mentioned earlier, JSF can be combined with different presentation layer technologies, so the details of what the template looks like may vary depending on the options supported by the JSF implementation. For better or for worse, JSF 1.0 requires all implementations to support JSP as one of the possible presentation layer technologies. On one hand, JSP is familiar to many developers, so it lowers the barrier of entry to JSF. On the other hand, JSP has its own mechanism for adding dynamic content to a static template; when mixed with JSF UI components, there's a risk for confusion and clashes between the two technologies. I use JSP for the newsletter application and for most of the other examples in this book, but you should be aware that it's not the only option. Don't worry too much about the potential clashes between JSP and JSF. I'll explain the issues and show you how to steer clear of the potential problems as we encounter them.

When JSP is used as the presentation layer technology, the page author creates a JSP page with the static content plus special elements that represent JSF components. Example 2-1 shows a JSP page with JSF elements for the newsletter subscription form.

Example 2-1. JSP page with JSF elements for the subscription form (newsservice/subscribe.jsp)

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

Example 2-1. JSP page with JSF elements for the subscription form (*newsservice/subscribe.jsp*)

```
<title>Newsletter Subscription</title>
</head>
<body>
  <f:view>
    <h:form>
      <table>
        <tr>
          <td>Email Address:</td>
          <td>
            <h:inputText value="#{subscr.emailAddr}" />
          </td>
        </tr>
        <tr>
          <td>News Letters:</td>
          <td>
            <h:selectManyCheckbox value="#{subscr.subscriptionIds}">
              <f:selectItem itemValue="1" itemLabel="JSF News" />
              <f:selectItem itemValue="2" itemLabel="IT Industry News" />
              <f:selectItem itemValue="3" itemLabel="Company News" />
            </h:selectManyCheckbox>
          </td>
        </tr>
      </table>
      <h:commandButton value="Save"
        action="#{subscrHandler.saveSubscriber}" />
    </h:form>
  </f:view>
</body>
</html>
```

At the beginning of Example 2-1, you find two JSP custom tag library declarations. If you're not familiar with JSP, don't worry. I'll explain what you need to know in Chapter 4. For now, just accept that these declarations identify all elements with the specified prefixes as special JSF elements. Elements with the prefix `h` (short for HTML) represents the standard JSF UI components combined with HTML renderers; elements with the prefix `f` (short for Faces) represent validators, event listeners, etc. that can be attached to the UI components.

Following the custom tag library declarations, there are HTML elements for layout and JSF elements corresponding to the JSF UI components. Ignoring the `<f:view>` element for now, the first JSF element of interest is the `<h:form>` element, representing a JSF form component. The same as in HTML, JSF input components must always be nested within a form component.

The email address input component is represented by the `<h:inputText>` element. The `value` attribute contains a value binding expression that binds the component to the `emailAddr` property of an application bean named `subscr`. As you may recall, this is the name assigned to the `Subscriber` bean in the JSF configuration file. The list of newsletter choices is represented by an `<h:selectManyCheckbox>` element with a

nested `<f:selectItem>` element for each choice. The `<h:selectManyCheckbox>` element value attribute is set to a value expression that binds the component to the `subscriptionIds` property of the `Subscriber` bean available through the `subscr` variable.

Finally, there's an `<h:commandButton>` element representing the Save button, with an `action` attribute containing a method binding expression that binds it to the `saveSubscriber()` method of the `SubscriberHandler` bean available through the `subscrHandler` variable.

Figure 2-3 illustrates what happens when a user requests the JSP page for the first time, and how the value and method binding expressions for the JSF elements in the JSP page combined with the bean declarations in the `faces-config.xml` file ties the whole thing together.

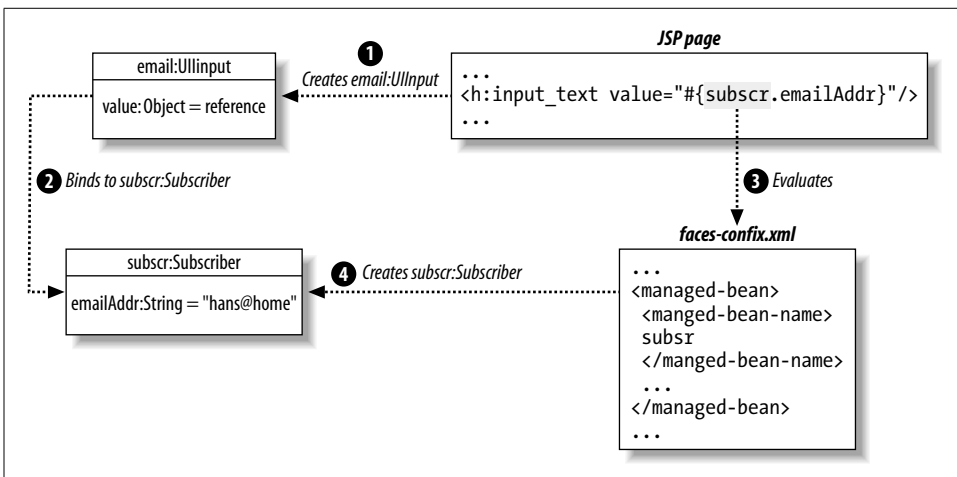


Figure 2-3. Object creation when processing the JSP page the first time

When the `<h:textInput>` element is processed, the corresponding `Ullinput` component is created and bound to the bean property specified by the value binding expression. The component is then asked to render itself. It evaluates the value binding expression and, if the bean doesn't already exist, JSF creates it based on the information in the `faces-config.xml` file. The input component pulls its value from the property of the bean specified by the rest of the value binding expression and uses it as the value of the HTML `<input>` element it renders. The other JSF elements are processed in the same way, and the combination of static content and the content generated by the JSF components is sent to the browser.

When the user enters values in the form and clicks the Submit button, JSF processes the request by asking each component to get its value from the request. Each input component sets the bean property it's bound to, and the command component fires an event that causes the method it's bound to be invoked. The method typically

saves the new values to a database or does some other backend processing, but in this simple example, as you may recall, it just writes the values to the console.

JSF then creates a response by processing the same or another JSP page (depending on the event processing method's return value and configuration-options we haven't talked about yet). If the same page is used, the component and application objects already exist, so no new objects are created. Other than that, the JSP page is processed exactly as for the first request.

This is a very simplified description of what really occurs at runtime. As you will learn, there's a lot more that may happen when a form is submitted, such as input value conversion and validation, error message queuing, forced redisplay of the same page without updating any bean properties, and more. But I've tried to keep it simple in this chapter to help you understand the basics of how the application code and the JSF components fit together. If you're still feeling dazed and confused after this whirlwind tour through the JSF architecture and the request processing, that's okay. Things will clear up when we go through all of this again, step-by-step and in more detail in the following chapters.