Introducing JavaServer Faces

JavaServer Faces (JSF) has been dubbed the next big thing in Java web programming. With JSF, you use web components on your web pages and capture events caused by user actions. In the near future, Java tools will support this technology. Developing web applications will be similar to the way we write Swing applications today: dragging and dropping controls and writing event listeners. This article is an introduction to JSF. It highlights the most important aspect of JSF: JSF applications are event-driven. Also, it offers a sample JSF application that illustrates the event-drivenness of JSF. To understand this article, you need to be familiar with servlets, JSP, JavaBeans, and custom tag libraries.

First of all, a JSF application is a servlet/JSP application. It has a deployment descriptor, JSP pages, custom tag libraries, static resources, et cetera. What makes it different is that a JSF application is event-driven. You decide how your application behaves by writing an event listener class. Here are the steps you need to take to build a JSF application:

- Author JSP pages, using JSF components that encapsulate HTML elements.
- Write a JavaBean as the state holder of user-input and component data.
- Write an event listener that determines what should happen when an event occurs, such as when the user clicks a button or submits a form. JSF supports two events: `ActionEvent` and `ValueChangedEvent`. `ActionEvent` is fired when the user submits a form or clicks a button, and `ValueChangedEvent` is triggered when a value in a JSF component changes.

Now, let's take a look at how JSF works in detail.

JavaServer Faces Technology Benefits

One of the greatest advantages of JavaServer Faces technology is that it offers a clean separation between behavior and presentation. Web applications built using JSP technology achieve this separation in part. However, a JSP application cannot map HTTP requests to component-specific event handling nor manage UI elements as stateful objects on the server, as a JavaServer Faces application can. JavaServer Faces technology allows you to build web applications that implement the finer-grained separation of behavior and presentation that is traditionally offered by client-side UI architectures.

The separation of logic from presentation also allows each member of a web application development team to focus on his or her piece of the development process, and it provides a simple programming model to link the pieces. For example, page authors with no programming expertise can use JavaServer Faces technology UI component tags to link to server-side objects from within a web page without writing any scripts.

Another important goal of JavaServer Faces technology is to leverage familiar UI-component and web-tier concepts without limiting you to a particular scripting technology or markup language. Although JavaServer Faces technology includes a JSP custom tag library for representing components on a JSP page, the JavaServer Faces technology APIs are layered directly on top of the Servlet API, as shown in This layering of APIs enables several important application use cases, such as using another

presentation technology instead of JSP pages, creating your own custom components directly from the component classes, and generating output for various client devices.

Most importantly, JavaServer Faces technology provides a rich architecture for managing component state, processing component data, validating user input, and handling events.

How JSF Works

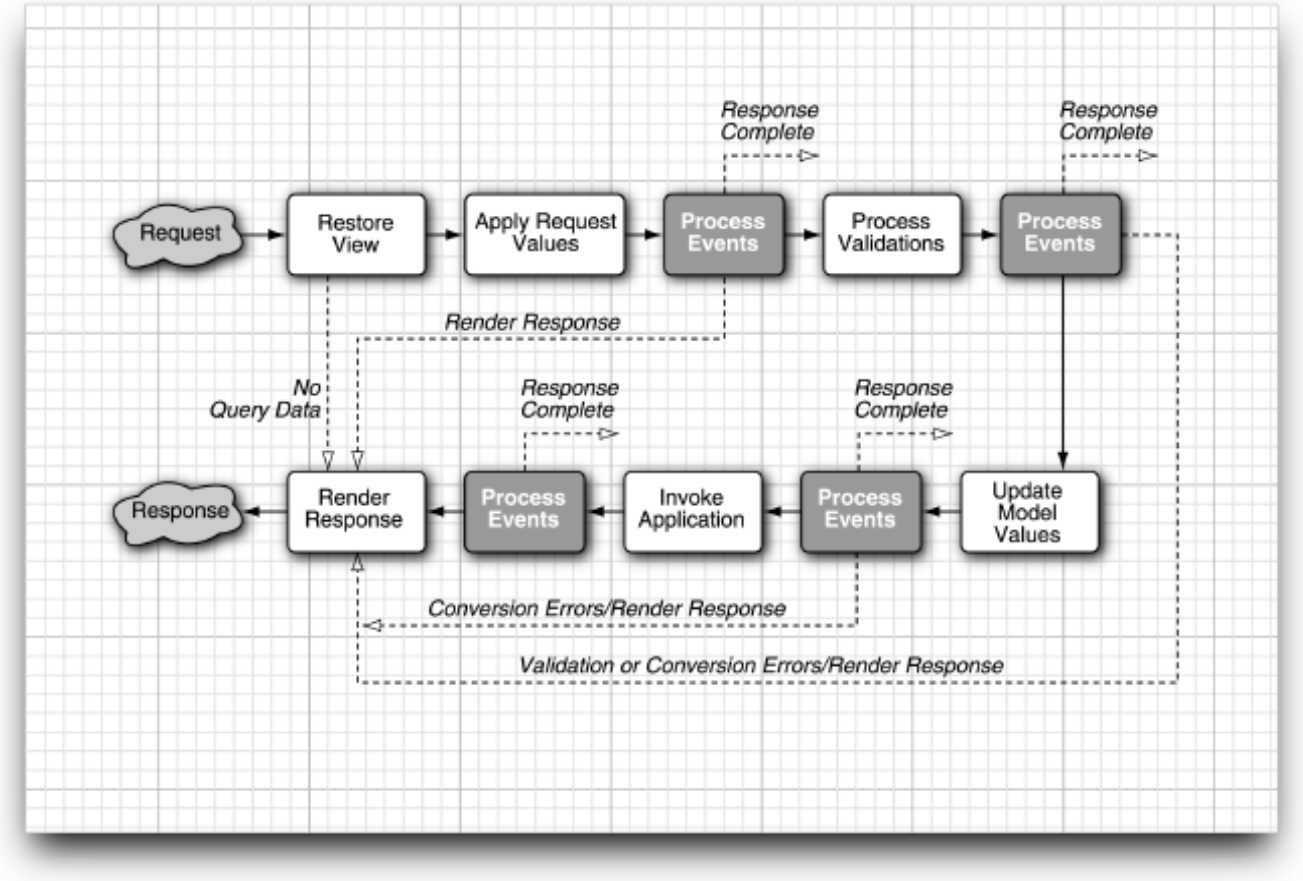
JSP pages are the user interface of a JSF application. Each page contains JSF components that represent web controls, such as forms, input boxes, and buttons. Components can be nested inside of another component; an input box can reside inside a form. Each JSP page is represented by its component tree. JavaBeans store the data from user requests.

Here is the interesting part: every time the user does something, such as clicking a button or submitting a form, an event occurs. Event notification is then sent via HTTP to the server. On the server is a web container that employs a special servlet called the Faces \servlet. The Faces servlet, represented by the`javax.faces.webapp.FacesServlet` class, is the engine of all JSF applications. Each JSF application in the same web container has its own Faces servlet. Another important object is `javax.faces.context.FacesContext`, which encapsulates all necessary information related to the current request.

The Life Cycle

The JSF specification defines six distinct phases.

1. **Restore View phase**
2. **Apply Request Values phase**
3. **Process Validations phase**
4. **Update Model phase**
5. **Invoke Application phase**
6. **Render Response phase**



As shown in the above figure, the normal flow of control is shown with solid lines; alternative flows are shown with dashed lines.

The
Restore View phase retrieves the component tree for the requested page if it was displayed previously or constructs a new component tree if it is displayed for the first time. If the page was displayed previously, all components are set to their prior state. This means that JSF automatically retains form information.
For example, when a user posts illegal data that are rejected during decoding, the old inputs

are redisplayed so that the user can correct them.

If the request has no query data, the JSF implementation skips ahead to the Render Response phase. This happens when a page is displayed for the first time. Otherwise, the next phase is the
Apply Request Values phase. In this phase, the JSF implementation iterates over the component objects in the component tree. Each component object checks which request values belong to it and stores them.

In the
Process Validations phase, the submitted string values are first converted to “local values,” which can be objects of any type. When you design a JSF page, you can attach validators that perform correctness checks on the local values. If validation passes, the JSF life cycle proceeds normally. However, when conversion or validation errors occur, the JSF implementation invokes the Render Response phase directly, redisplaying the current page so that the user has another chance to provide correct inputs.

After the converters and validators have done their work, it is assumed that it is safe to update the model data. During the
Update Model phase, the local values are used to update the beans that are wired to the components.
In the
Invoke Application phase, the action method of the button or link component that caused the form submission is executed. That method can carry out arbitrary application processing. It returns an outcome string that is passed to the navigation handler. The navigation handler looks up the next page.
Finally, the
Render Response phase encodes the response and sends it to the browser. When a user submits a form, clicks on a link, or otherwise generates a new request, the cycle starts anew.
You have now seen the basic mechanisms that make the JSF magic possible. In the following chapters, we examine the various parts of the life cycle in more detail.

JSF and Ajax

JSF is often used together with Ajax, a Rich Internet application technology. Ajax is a combination of technologies that make it possible to create rich user interfaces. The user interface components in Mojarra (the JSF reference implementation[7]) and Apache MyFaces were originally developed for HTML only, and Ajax had to be added via JavaScript. This has changed, however:

Because JSF supports multiple output formats, Ajax-enabled components can easily be added to enrich JSF-based user interfaces. The JSF 2.0 specification provides built in support for Ajax by standardizing the Ajax request lifecycle, and providing simple development interfaces to Ajax events, allowing any event triggered by the client to go through proper validation, conversion, and finally method invocation, before returning the result to the browser via an XML DOM update.