JSR127 JavaServer Faces

by **Bart Leeten (EDS)** & **Kris Meukens (EDS)**

Not many Java Specification Requests (JSRs) have been that much anticipated and contested as JSR127 JavaServer Faces (JSF):

- Does one really need anything beyond JavaServer Pages (JSPs) and Servlets?
- Aren't there already enough web application framework offerings?
- Isn't Struts already filling the gap as the de-facto web application framework for the Java 2 Enterprise Edition (J2EE) platform?
- Should the Java Community Process (JCP) not invest its time and efforts in evolving/converging the already existing frameworks instead of reinventing the wheel?
- J2EE will never be able to catch up again since Microsoft's smashing .Net WebForms and the escorting tool support in Visual Studio.NET, so why still bother about the web presentation tier?

This article will not only provide you with an overview of what JavaServer Faces is about, but also with the arguments why we believe JavaServer Faces really presents a significant step forward for the J2EE Web Tier. As JSF is still work in progress, not cast in stone yet, we will limit ourselves to describing the overall features and goals, and will not overwhelm you with lower level Application Programming Interface (API) details.

For clarity reasons and to stress that JavaServer Faces is not only about 'visual' user interfaces, we propose to use the term 'face', to express what for visual interfaces is typically named a 'screen'.

Observations

When observing the construction of web user interfaces on the J2EE platform, one can notice that before getting the development team truly productive, one first has to deal with a considerable number of common 'infrastructure' issues.

Therefore development teams will often attempt to:

- Abstract away all the plumbing required to process user actions, for example event handling, validation, error handling, navigation from *face* A to *face* B, model state management, internationalization, localization, web accessibility, … by the creation or usage of a web application framework.
- Remove, often duplicated code in JSPs, required to create a *face* (e.g. the mark-up and accompanying script functions that make up a tree widget) in custom tag libraries. Often widgets are created so that they correctly render themselves for different user agents, user agent versions, etc. For that matter, web application frameworks frequently include user agent capability sensing.
- Define rigid guidelines to which developers have to comply in order to be consistent with an agreed look and feel.
- Speed up the development of *faces* by automating the coding process as much as possible i.e. code generation based on meta data, wizards, templates, …

Think about how many proprietary decisions and costly assemblies have to be/are made before one can even start to think about what really matters: designing a 'quality' application addressing the business requirements, on time and within budget.

That was the reason why in May 2001 a significant number of players within the JCP joined forces to form an expert group with the goal of defining a standard user interface (UI) toolkit, called JavaServer Faces. Moreover the expert group will make JSF:

- leveragable by (off-the-shelf) development tools
- integratable with and built on top of standard J2EE technologies like servlets, JSPs, tag libraries (like JavaServer Pages Standard Tag Library (JSTL))
- integratable with new (standard) specifications as portlets (as they are currently being defined in JSR 168).

In fact it is envisioned that JSF becomes a standard part of the J2EE platform.

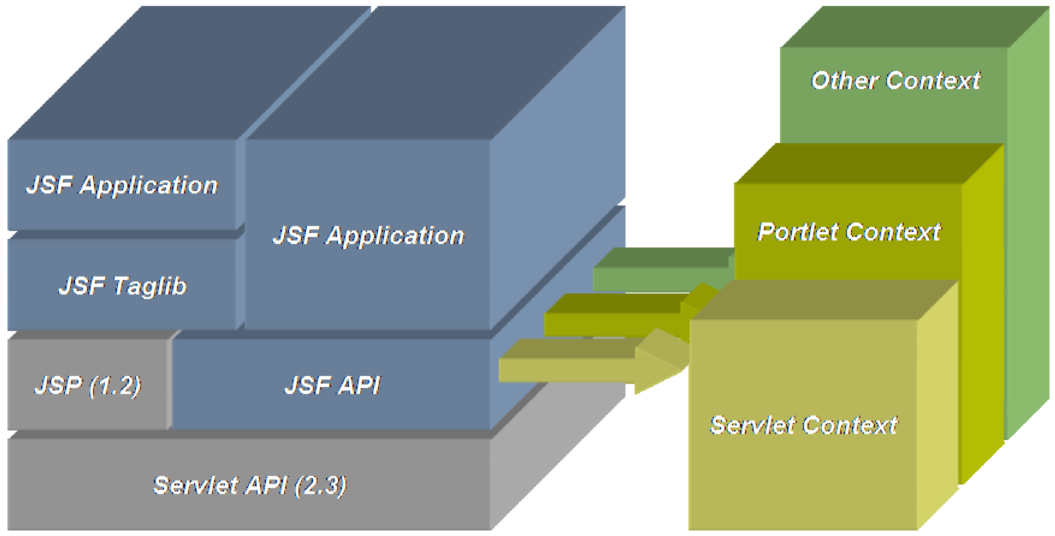
On the other hand the JSF expert group does not want to ignore existing frameworks. Firstly the expert group recognizes valuable concepts and patterns applied by existing frameworks and cherry picks from them. Secondly they are assuring that JSF has integration points with other frameworks. What the expert group actually tries to accomplish is to standardize the web application framework API. That way framework builders can concentrate on creating the best possible JSF implementation while the developers get not only a standardized API code to code against but also (visual) tool support and pluggable JSF implementations. As such JSF presents a preservation of current investments and a smooth migration path away –when applicable - from a proprietary framework. JSF will be a stimulant for web application framework developers to converge towards standard (J2EE) APIs.

A good example of a migration/integration strategy is the JSF integration library for Struts 1.1 (see http://cvs.apache.org/viewcvs/jakarta-struts/contrib/struts-faces/) and the plan to replace in the Struts tag library by JSF tag library in Struts 2.0 (see http://jakarta.apache.org/struts/status.html).

Under the hood

Let's have a brief look on how JSF delivers its functionality. JSF basically builds upon the web tier APIs available within J2EE 1.3 i.e. JSP 1.3 and the Servlet 2.3 API, hence it can be used in combination with all the major application servers today available on the market.

Most of the time the containing environment of a JSF application will be accessible via the Servlet API, however as JSF can also be used within other containing environments e.g. in the context of Portlets it has to cater for this possibility. In fact the inner-workings of JSF are shielded from their external context and kept as 'pure' as possible, this assures JSF's growth path towards the future.



Modi operandi

JSF supports two modi operandi: with or without JSPs. For use with JSPs, a JSF implementation must provide at least two JSF tag libraries. They sit, as a thin layer, on top of the JSF API. The tags can be embedded in JSPs, very similar to how user interfaces are typically built with Struts for example. The main tag library, also known as the jsf-core tag library, contains render independent JSF tags. Other tag libraries offer render(kit) dependent tags.

Nevertheless there are some shortcomings inherent to JSPs – e.g. JSPs are often render-dependent - and in order to address those, JSF offers developers direct access to the JSF API skipping JSPs and tags.

This can be very practical when you for example want to produce *faces* that require extremely flexible (e.g. not only to be rendered in HTML but also in SVG) and adaptive layouts (e.g. resizable elements). In that case developers will be able to apply absolute positioning or layout management techniques similar as in Swing/AWT/SWT, to provide a flexible programmatic mechanism to lay out *faces*. The composition of such a *face* can be performed with tools similar to the ones currently available for the creation of Swing/AWT/SWT-based user interfaces. This is a major step for the J2EE tool market to catch up with and leapfrog the capabilities that are currently offered by Microsoft's VisualStudio WebForms capabilities in that regard.

Component and Render Kits

Another analogy between JSF and Swing/AWT/SWT is that JSF is also component based, meaning that under the hood every part of a *face* is represented as a tree of UI components. Similar to Swing/AWT/SWT the component model used is render technology agnostic. Its naming scheme is inspired by W3C's XForms. In other words the developer uses a generic component model that at run-time will delegate the actual mark-up rendition to specialized objects called renderers.

Example: a UICommand can visually be represented as an HTML hyperlink, an HTML button or even a rich SVG button. The concern how it will be rendered is separated from the actual 'under the covers' generic processing. In order to switch the presentation from a hyperlink to a button, the developer just has to associate another renderer type. In JSPs, the renderer type is indicated via the embedding of a render type specific tag (e.g. the reference implementation's tag library offers the tags <UICommand_Hyperlink/> and <UICommand_Button/>).

Render kits can be knowledgeable about the user-agent's capabilities and/or the user's preferences. Initially (in JSF 1.0) render kits will be an inseparable collection of renderers designed to optimally work together. However render kits will not be the standard mechanism to create different skins or look & feels. The standard mechanism that provides that functionally is deferred to a later release. It turns out to be very complex to design a solution that is really render independent, W3C's Cascading Style Sheets (CSS) are not the all-inclusive solution to skin widgets.

Render kits could be selected according to the capabilities sensed by an advanced JSF implementation, an elegant mechanism to do so could be through the capabilities offered by JSR188 CC/PP Processing. The latter JSR's expert group is defining a standard capabilities sensing API towards standards as W3C's Composite Capability/Preference Profiles which is one of the common mechanisms to capture user-agents capabilities. Notice that Render kits are not necessarily limited to rendering HTML (in combination with possible scripting and/or CSS). On the contrary component rendering in for example SVG, PDF or Flash MX, is perfectly within reach. As JSF 1.0 render kits will offer an inseparable set of renderers, render kits will typically concentrate on a single content type.

Any JSF implementation must at least feature an HTML 4.01 render kit. In fact a whole 3rd party market, offering specialized components and/or render kits, is on the verge of emerging. Soon , JSF components and/or render kits will be available as currently already exist for some time for Microsoft's ASP.NET.

Model Beans

The above UI components and render kits are one part of the story. An other part is the model, realized with the so-called 'model beans' that hold the information to be presented by UI components. These JavaBeans' compliant model beans are registered for a particular duration of time which can be the lifespan of a request, a session, the application or more simply the bean can be recreated every time it is required. Registration can happen in 2 ways: programmatically or through the use of an XML configuration file. During this registration process, model beans can be instantiated and also be populated with information. This information can be statically defined or dynamically retrieved from elsewhere. A JSF implementation can offer two mechanisms to retain model bean state over time: on the server (which is the default) or optionally on the client. The latter requires the model bean state to be serialized into the rendered response and therefore is dependent on the render kit's capabilities in that sense. The inclusion of that capability could be a discriminating feature in your render kit implementation selection.

Events

A *face* is not limited to representation, it also offers the ability to interact with the information through the usage of the appropriate controls. These actions will result in events, that will be handled by their respective listeners.

Remark that events can originate from both external stimuli and JSF internal workings. An illustration of this is a user clicking on a tree-node to expand a tree. That action could create a server-side event setting off a registered listener to retrieve not-yet retrieved leaf information.

In order to define an unambiguous request processing lifecycle that indicates what needs to happen when, that lifecycle is divided into phases. Currently there are 6 phases defined each with its distinct responsibility: *reconstitute tree*, *apply request values* (*), *process validations* (*), *update model values* (*), *invoke application* (*) and *render response*. The phases indicated with (*) include a shared sub-phase called event processing (this is a significant change with older JSF early access releases). Therefore event listeners need to indicate during their registration process in which phase they need to be notified.

Validators and Converters

During the *process validation* phase dedicated helper objects come into play namely 'validators' and 'converters'. A validator will validate for example the value entered in a UIInput component and when necessary register 'message' objects that can be used by the application to show error messages. The contents of those messages can be retrieved from resource bundles in order to enable localization of those messages. Remark that validators are not limited to the validation of the contents of a single component.

For example a particular validator can be responsible to perform the validation of a bank account number that is represented by a custom bank account number component composed out of 3 separate UIInput components.

Converters are used to transform the data as it is represented on the user interface into the type as it is required within the model bean. A typical example is a date that is represented on the screen as the string '24/01/2003' but internally stored in a model bean that is an instance of the type 'java.util.Date'.

Wrap-up

Let us in a few words re-iterate this article's key points:

- JSF will protect investments as it
    - Can be integrated with your existing web application framework
    - Will become part of the standard Java (2EE) platform APIs
    - Is not a product but an API, thus providing choice of provider/implementation
    - Offers developers a single programming model independent of user agent and channels through its abstract user interface component model.
- JSF will improves flexibility:
    - Thanks to the unrelenting strategy of separating concerns.
- JSF will be an enabler by
    - Forming the foundation of a new market for JSF implementations, components, render kits and development tools.
    - Facilitating more developers to create more sophisticated (web) applications thanks to its implementation encapsulation.
    - Raising web user interface construction on the J2EE platform to new levels. It can be used both with and without JSPs and is not limited to visual user interfaces.
- JSF will bring quality improvements by
    - Enabling developers to plug-in proven, pre-tested implementations and render kits thanks to the standardized JSF API.
- JSF will increase productivity
    - Through its ability to be leveraged by tools
    - By offering a render independent component model so that developers work with one single model independent of the applied render technology.

- By encapsulating the complexity involved with the development of rich, complex user interface.

To conclude we mention that by the time you read this article JavaServer Faces will be available as public draft 2. The final release of version 1.0 is targeted for Q3 of 2003.

For more information you can always contact us by email kris.meukens@eds.com or bart.leeten@eds.com or find more information at http://java.sun.com/j2ee/javaserverfaces/ and http://www.jcp.org/en/jsr/detail?id=127.

Bio:

Kris Meukens and Bart Leeten

- Working with Java technology since 1997, primarily in the automotive, financial and public industries throughout Europe
- Consult and coach EDS customers on the usage of object, web, Java, XML and integration technology for the development of large enterprise information systems
- Consult EDS customers on ‘electronic product’ (software & other) lifecycle management
- Kris represents EDS within the JSR 127 (JSF) and JSR 168 (Portlets).
- Bart represents EDS within the JSR 127 (JSF) and JSR 152 (JSP 2.0).