

# U, I AND JSF



27.07.2009

JSF – still painful after all these years  
by Karl Banke

*Back in 2002 I wrote a little piece about what I did not like about JSF 1.0. Since then I have been doing development in the web domain using various home grown frameworks (both our own and those of customers), I have the switched to work more on the server side, designing services as well as working on Java Swing Thick Client Applications. Recently I came back to use JSF at a client's site. I was hoping for the best. But it turned out that there is still a lot of stuff I do not like about JSF. Actually, the more I work with it, the more worked up I get.*

# U, I and JSF

STILL PAINFUL AFTER ALL THESE YEARS

## Inhalt

ABOUT THE AUTHOR .....	1
THE SCENARIO .....	1
INITIALIZATION.....	2
RE-INITIALIZATION .....	3
A Tomahawk subtlety .....	3
FINDING COMPONENTS.....	4
VALIDATION.....	4
STYLING.....	5
MIX AND MATCH COMPONENTS.....	6
FACELETS TO THE RESCUE.....	7
CONCLUSION .....	7
REFERENCES .....	8

## ABOUT THE AUTHOR

Karl is a senior Java architect and developer. He has been working as a consultant in the J2EE space for more than 10 years. He is one of the authors of the groundbreaking book "Enterprise SOA" published in 2004. He also participated in the development of the iternum UI framework at his own company. He is married without children. You can reach him at [karl.banke@iternum.com](mailto:karl.banke@iternum.com).

## THE SCENARIO

The project where I had to cope with JSF again, after years of absence, is based on a product called BEA WebLogic Portal 9.2. This is a JSR-168 Portlet Server running on top of WebLogic Server 9.2, a standard J2EE 1.4 container. This of course puts some limits on the version of JSF I could use. JSF 1.2 won't work, since it requires JEE5. That said, JSF 1.2 adds very little to the specification to ease my pain.

JSF 2.0, which does address some – yet by n means all – of my biggest issues with JSF is totally out of the question, since it requires JEE6 and the specification has not been finalized until very recently. So this is to be considered more a report from the trenches of real world coding rather than an academic discussion about UI frameworks. In our scenario we use the Myfaces JSF implementation together with the Tomahawk extensions which is a fairly common setup.

There is one sideline as well – we do not want to develop any custom JSF components if we can at all help it. The reason is that this will likely deviate from the standard and will make upgrading to a newer version of JSF, JEE and the portal server in particular rather painful.

Most of the problems I have with JSF are well known to a certain extent. The odd thing is that the open source community has only very little solutions for these problems other than “roll your own component”. So let’s investigate some of the really sore spots of JSF one by one.

## INITIALIZATION

Initialization is a particular problem of JSF. The problem here is that JSF does not have a standardized and usable way of dealing with unexpected error conditions. The problem is fairly simple. Consider a JSF Page that displays a table of values that are read from a database. Using JSF you will provide this data via a backing bean and expose a collection of rows. So you will do something like this.

```
<f:view>
<h:form>
<t:dataTable
    value="#{myBacking.persons}" var="row"
    styleClass="scrolltable">
    <t:column>
        <f:facet name="header">
            <h:outputText value=" " title="Name"/>
        </f:facet>
        <t:outputText value="#{row.name}"/>
    </t:column>
    ...
</t:datatable>
</h:form>
</f:view>
```

Now what happens if the getter fails that is, if the `getPersons()` method of the backing bean throws an exception. There is in fact very little one can do. You can use the standard JSP error mechanism but that’s about it – and it is not what you would generally want in particular in a portal environment. Instead you would want to display a particular error message in the context of the particular JSF application – in JSF this would be done by navigating to a different view where you would be told something like “The person database is currently not available”. Unfortunately, when rendering this page for the first time this is just impossible, since the rendering has started before `getPersons()` even gets called. So the only way to safely accomplish such a thing is to do it programmatically within the JSP page before the rendering has actually started.

```
<%
    FacesContext facesContext = FacesContext.getCurrentInstance();
    Application application = facesContext.getApplication();
    ValueBinding binding =
        application.createValueBinding("#{myBacking}");
    PersonBean bean = (PersonBean) binding.getValue(facesContext);
    if (bean.isInitRequired) {
        bean.init();
    }
%>
<t:dataTable rendered="#{myBacking.ready}"
...
</t:datatable>
<t:panelGroup rendered="#{!myBacking.ready}">
<!-- Error content goes here -->
...
</t:panelGroup>
```

Somewhat ugly isn't it. Yet this is the only save way of doing something like custom error handling in JSF.

## RE-INITIALIZATION

This is almost as hard as initialization itself. Sometimes it is necessary to reinitialize an entire JSF page. The obvious way would be to get hold of the UIRoot in the Action Phase and just discard the component tree. The problem is that there is no way to accomplish this – at least none that works to my knowledge.

Another way of re-initialization is of course to create bindings to the components and init them programmatically. However it can be rather tiresome to reset table indices, switch tabs and so on just to simply start over. Yet for all practical purpose, it is the only way to restart entirely from scratch.

### A Tomahawk subtlety

Reinitialization of simple component values can also have strange results. Consider the example below. Upon submission, the values in the backing bean are reset prior to rendering. While the textfield that is not in the subform can be reset directly, it is impossible to reset the field in the same way containing the input in the extremely handy subform tag. In this case arguably you can't blame JSF, since the subform tag is an extension to JSF in Tomahawk. However this is extremely annoying simply because the subform component is so useful.

```
<f:view>
<h:form>
  <t:outputText value="#{SimpleViewBean.messageValue}"/>
  <br/>
  <t:outputLabel for="input1" value="String"/><t:inputText
id="input1" value="#{SimpleViewBean.stringValue}"/>
  <br/>
  <t:subform id="subform">
    <t:outputLabel for="input2" value="Number"/><t:inputText
id="input2" value="#{SimpleViewBean.longValue}" required="true"/>
  </t:subform>
  <br/>
  <t:commandLink value="Submit" title="Execute"
action="#{SimpleViewBean.reinit}"/>
</h:form>
</f:view>
```

```
public class SimpleViewBean {

  private String stringValue = null;
  private String longValue = null;
  private String messageValue = null;

  public void reinit() {
    messageValue="You entered " + stringValue
      + "and" + longValue;
    stringValue=null;
    longValue=null;
  }

  public String getLongValue() {
    return longValue;
  }
}
```

```
public void setLongValue(String longValue) {
    this.longValue = longValue;
}

public String getStringValue() {
    return stringValue;
}

public void setStringValue(String stringValue) {
    this.stringValue = stringValue;
}

public String getMessageValue() {
    return messageValue;
}

public void setMessageValue(String messageValue) {
    this.messageValue = messageValue;
}
}
```

## FINDING COMPONENTS

Due to some shortcomings mainly in validation and initialization, you may need to access components in your backing bean action methods quite frequently. One way is to wire each component you need to a field in your backing bean which will clutter code unnecessarily. It would be much better if there would be a standard way to find a component by name. In fact JSF's `UIComponent` has a method called `findComponent`. Unfortunately, this method stops searching the component tree whenever it encounters a naming container when a search is performed.

What seems like a good idea at first sight turns out to be unexpectedly painful in the real world. The reason being that it is not that transparent which components create their own naming container – and it is easy to forget and can cause really a lot of frustration, in particular when a piece of JSF code along with its Managed Beans is supposed to be used in a different context.

It would be much easier – and trivial – if an addition to the `findComponent()` method there would be an overloaded method that returns all components that match a particular name regardless of the `NamingContainer` they are in.

## VALIDATION

You will find a lot of content on the internet about the drawbacks of JSF validation. Its main problem is that you cannot validate fields conditionally. Basically, JSF will only allow you to validate one component at a time, one after the other. If a conversion or validation error occurs, JSF will skip all action processing and redisplay the initial page. This is such an impractical and outright stupid validation strategy that you are wondering what the hell was on the mind of the original authors of the specification. That some field in a form is required or has a certain format based on some other field is such a common use-case that not catering for it is pure negligence.

Examples? How about postcode requirements and validation based on the country selected? State required based on country? Different phone number formats for different countries? Format of account numbers based on country?

A bunch of fields being required based on a checkbox selected? Format checks for credit card numbers based on card type?

In JSF 1.1 and 1.2 there are strategies to accomplish this kind of behaviour, but they are very cumbersome. For example you can put all your validation in your actions and validate purely programmatically. Not doing something like this is precisely one of the reasons why you are using a component framework in the first place.

Or you can write a custom component that aggregates other components (the ones that you really want to validate) and create a custom validator for these. Strange as well – why create a component for no other reason but to provide validation hooks?

Or you can write a custom component encompassing all your relationships which is about the dumbest strategy I can think of, yet it is one that is often recommended by the “experts”. In the example above, based on the value of the country dropdown, I run off to create a component for country, phone number, state and postcode? I think not!

More than half a decade after this moronic validation scheme has been derived, JSF 2.0 is here to the rescue, finalized on July 1<sup>st</sup> 2009 (I did mention I will not be able to use that for the foreseeable future, did I?). Or is it? Looking at the specification it now supports something called bean validation and default validators. Useful, but not what I want from a UI framework. I want validators that act on and are aware of more than one field!

## STYLING

Back when JSF 1.0 was derived, though the AJAX wave was still quite some time to the future, it was crystal clear that CSS was the standard for creating a consistent look and feel for styling web applications. Given this simple fact, I am still starstruck as to how poorly the JSF specification deals with CSS integration. The basic JSF Tags allow for integration of CSS styles and classes for the individual components. So you can do something like this for a textfield and its label.

```
<t:outputLabel styleClass="label" for="name"
value="Name" tooltip="Enter last name"/>
<t:inputText styleClass="longTextStyle" id="name"/>
```

Does not look bad on first sight, does it? Unfortunately, it is more or less totally worthless. Neither in the specification nor in the tag library, there is any way to figure out the state of a component. If a component becomes required for some reason after creation of the component tree, there is no way to apply a different style. Worse still, the label tags (and components) do not expose any properties of their linked components. Whoever thought of this super encapsulation strategy should be encapsulated some place where he or she will not be able to mess around with future specifications. Do you think I am unfair? Let me give you a bit of an outline, what cannot be done with the default JSF specification.

- Cannot render label style based on input field being required
- Cannot render label style based on input field being invalid
- Cannot render label tooltip based on input being invalid
- Cannot render field style based on input being invalid
- Cannot render field tooltip based on input being invalid

In total, there is no support at all for changing rendering styles conditionally based on properties of some other component or indeed based on the component itself. The shameful thing about this is that behaviour

like that is Web One-O-One since approximately 1995. Even worse, there is not even an API that would allow for performing these tasks programmatically in a somewhat decent manner.

And support could be added in very simple ways. For example one could expose a variable containing the component itself or the referenced component to JSF EL and make sure the relevant properties are evaluated for each page execution. Or one could bite the bullet and define a general purpose CSS standard for skinning components and provide a decent basic render kit.

There is also some layout support in the JSF specification. This however is so utterly poor that one wonders why the hell is it there in the first place. The JSF Tags PanelGroup and PanelGrid can be used to this extent. I vaguely remember that by 1999 it was pretty clear that a HTML table might not be the best layout mechanism. That did not stop those experts to hard code the PanelGrid using tables and of course not provide an option of replacing it with a more up to date mechanism.

One would think JSF 2.0 could deal with these easy enough to solve issues. But it does not does much about it. This is perfectly understandable, since most frameworks that are based on JSF have added some support for the issues mentioned above. In other words, people have tackled the problem using different strategies and are not inclined to standardize their component sets properly. Most of the products based on JSF are not simple component libraries and are not envisioned to be used that way. Rather they are application frameworks adding their respective support for conversations, page scope handling, backend component integration and AJAX support – so in order to provide a bit of sensible styling you really need to go a long way.

And of course, there are side effects. As an example, when adding Seam's decorate Tag to the equation in order to obtain solve some of the problems mentioned above, your code may break since the Tag is itself a Naming Container...

## MIX AND MATCH COMPONENTS

One of the greatest promises of using a component framework like JSF is that you'd be able to mix and match components from various sources. I cannot see that JSF fulfils this promise to any relevant extent. While there are various implementations of JSF, like ADF, RichFaces or IceFaces they all seem to be fairly hard to get to work together. That is no fault of theirs. The reason is that – as outlined above – JSF is essentially pretty much underspecified for most practical purposes.

So in order to create nice components or even a nice working framework, people need to make assumptions that may or may not be compatible when combined. For example, IceFaces essentially JSF, but without its ajaxified partial form submit mechanism it is pretty useless. And of course, these postback facilities will not magically shine up on any other component I choose to drop into my IceFaces App.

RichFaces, another component library that is based around Ajax concepts is suffering from the same effects. Various frameworks created their respective implementations of conversations and continuations that may not interoperate. In particular, Javascript namespace and version clashes are almost inevitable when combining various rich component sets. And don't get me started about running more complex ajaxified JSF components within a Portal environment – you'll need something called a portlet bridge which is more often than not vendor specific.

Since the style support in JSF is non existent for most practical purposes, component vendors usually attack the issue within their render kits and component implementations. This will likely create different namespaces and naming conventions for different component vendors that all need proper maintenance.

In other words: The under-specification of JSF in its AJAX and CSS support will require you to write your components yourself or to resort to what is provided in some JSF framework – the first is not desired while the latter will create some form of lock in.

## FACELETS TO THE RESCUE

Java Server Faces is a complex framework that has some merits. One particular merit is the clear separation between Logic and View. Thus it is fairly easy to extend the view handling mechanism without changing the action processing system of a JSF application. One thing that people who were used working with things like Tiles were missing in JSF was a templating system. This gave rise to the Facelets specification, as of JSF 2.0 a required part of the specification.

Whenever I went and told people about the problems I had with JSF, I earned a sad look and a punchline like “Why don’t you use Facelets then”. True enough, facelets offers some easy ways of designing your own “components” by aggregating components into a template. The templating system is actually quite powerful, but it does not address the main shortcomings of the application. Facelets do not enforce a better or more concise CSS usage (though they might encourage it). Facelets can do precious nothing exposing the much needed properties of components, like validity and required fields. Facelets won’t magically wipe away Ajax interoperability issues. And Facelets cannot overcome the validation problems in JSF, because they are not routed in the presentation layer to start with.

## CONCLUSION

JSF is often mentioned as the standard framework for Java Web Component integration. This alone often makes it a piece in some corporate strategy. But of all that it is, it is not a proper component framework, let alone a complete one. It is intrinsically over engineered, yet critically underspecified. As such, it can serve as a powerful basis for a real component framework. Because of the critical parts missing it is essentially impossible to mix JSF components from various providers without including a lot of overhead and losing a lot of consistency. Frameworks like JSF should essentially make easy things easy and hard things possible. Yet JSF’s standard components, even in JSF 2.0, fail to deliver basic One-O-One web application features from 10 years ago. It is a remarkably complex specification that delivers pretty much nothing out of the box.

This is not to say that there are no good frameworks implemented with JSF. There are quite some. But if you are using them, essentially you won’t be doing JSF, but you’ll be doing ICE Faces or ADF and possibly blend these with other frameworks loosely based on JSF, like Seam. They may also offer excellent client side support – which is one more reason why they cannot interoperate properly.

You might also decide to create your own render kit or build your own components on top of JSF. There is really no reason why you shouldn’t rather than that this is not what a component framework should be for in the first place. The very reason for using a component framework is to use components that are there and build some of your own on a clean extension path. Instead you’ll end up writing most non trivial components yourself, and since JSF is a complex framework, writing your own JSF components is a lot of overhead.

Now you may ask the question: If JSF is as broken as you say, why don’t you do something and change it? Why don’t you, say engage more in the JCP and work towards it becoming a real component library. The brief answer to that is: I don’t do politics! The somewhat longer answer is: The JCP for JSF 2.0 consists of people who hold a strong investment in JSF as it is now. They are only to a certain extent inclined to standardize areas that they have fixed in their respective frameworks and libraries in different ways. Also, it is apparent that the JSF 2.0 specification had to aim for providing a common ground for AJAX integration, in particular in the areas of partial submit and partial rendering, in order to prevent the

specification to fall apart entirely. And finally the JSF 2.0 effort is closely tied into the JEE6 release cycle, thus only a limited number of desired features could be incorporated into this release to start with. But that does not mean that I may not phrase an opinion on the shortcomings of JSF – as have others and in big numbers for that matter. I just hope that for the next JSF release the experts will listen to the people – otherwise I fear JSF will be another technology to be voted down by people's feet eventually.

## REFERENCES

1. JSF 1.0 through 1.2 documentation is available online at <http://java.sun.com/javaee/javaserverfaces/reference/api/>
2. The JCP for JSF 2.0 is available at <http://jcp.org/en/jsr/detail?id=314>
3. The JSF for Nonbelievers series offers some excellent JSF tutorials [http://www.ibm.com/developerworks/views/java/libraryview.jsp?search\\_by=nonbelievers:](http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=nonbelievers:)
4. An interesting compendium of JSF rants and reviews <http://ptrthomas.wordpress.com/2009/05/15/jsf-sucks/>