Bookmarkable and SEO Friendly JSF with PrettyFaces

by Benny Bottema - Friday, March 20, 2009

http://www.bennybottema.com/2009/03/20/bookmarkable-and-seo-friendly-jsf/

Yes, it is possible. Yes, it requires hacking JSF with filters and/or phaselisteners and/or custom servlets for encoding/decoding serializing/deserializing GET parameters and force redirects upon JSF view resolvers. The good news however: it's all been done before and you can just hang back and use a small framework that does all this stuff for you. Enter PrettyFaces.

But before we come to that, I would like to show you how PrettyFaces sums up what you need. This involves solving two problems: **The JSF POST compulsion** and **the JSF Redirect problem**. You can in fact solve all of these problems separately, but PrettyFaces just encapsulates all these annoying things for you in a convenient little jar.

Contents

1. GET urls? The POST-back problem with JSF

In one sentence: JSF uses POST requests for *everything*, so there is are no nice copy-pastable GET urls. For example, if you use your generic h:commandLink, a POST submit is being done on the surrounding form to get everything into JSF. JSF needs this because the form contains a lot of extra (meta) data for the input to be processed by the JSF framework. You could for example indicate that the state information should be stored on the view instead of the server, which is then simply added to the HTML form behind the scene. Now this kind of stuff, you don't want in GET urls. It would cause every action you perform on the website to result in a hideous string of code in the browser bar, which isn't even copy/pastable to begin with since it contains encoded directions for the server based on specific state (ie. getEmailadresFromCurrentClient, which won't work if there is no current client available).

There are instances where you would want GET urls, which work in a stateless fashion. For example .../faces/newsarticle?article=2334. However, the developers at JSF apparently thought "If we don't need it, you don't need it", since there is no way in JSF to configure URL mapping using GET parameters.

And this is the heart of the problem. This is where all the hacks come in.

The POST-back solution

There are a couple of methods to approach this problem. Or rather, the concept is the same, but applied in different ways. The concept is that you intercept a HTTP request, manually extract the parameters from the request URL and apply them to your model (a JSF managed bean for example). The POST problem can be split up into two subproblems: GET url interpretation and GET url generation.

Solving the GET url interpretation

You can interpet GET urls in JSF by defining a custom Servlet and have these urls go through it rather than through the faces servlet. In the custom servlet, process the parameters and then redirect the browser to a faces url without these parameters. You need a redirect, since a server-forward won't trigger another servlet to act on the mapping. If you still want to forward on the server, you need to create a FacesContext, fix the viewroot and handle the view navigation yourself and so imitate what the faces servlet would've done. There are libraries out there that contain this solution ready to go.

You cannot do it with a filter for the Faces Servlet, because GET parameters will have been removed already. An alternative to GET parameters is to work the parameters into the url itself (something like http://www.peanutbutter.com/faces/newsitems/params/id-22234.xhtml), but this won't work either since a filter has no influence on the url passed on to the servlet for further processing. So you can't remove the url-based-parameters from the URL and let JSF deal with the rest; it will simply fail to understand our custom parameter format.

Another way that does work is to create a custom JSF PhaseListener. You'll still have lost the GET parameters, but at least you can do something about the url based parameters mentioned in the paragraph before (that and you already have a FacesContext you can use). In this case, you extract the parameters before the RESTORE_VIEW phase and use the resting url to manually set the view root for JSF to work with. Jason put an example of this manual url based parameter PhaseListener mechanism on his blog. Before PrettyFaces I based my own solution on this concept like Jason's, but Jason incorrectly comments [in his code] that you cannot infer the original view url in this case. I solved this by adding a keyword /params/ in between the url and parameters. Lucas from Amis has a variation on the manual url based parameter PhaseListener mechanism which interprets the parameters and then directly renders the views.

My own manual url based parameter PhaseListener maps the parameters to bean properties in an automated fashion (although the bean names, parameters and properties are hardcoded in a static list at the top, which could've come from an xml file or something making this version completely generic -which incidentally is exactly what PrettyFaces does-).

Solving the GET url generation

You can't have GET urls for every state in the application, that would require massive GET urls filled with everything JSF needs to restore its state. What you need are specific cases where you want your users to be able to use a GET url, such as viewing a specific news item, company information, always

things based on some identification. For these kind of GET urls, you can write a custom ViewHandler in JSF, which intercepts the view resolution and manually appends the GET parameters. For this to work, you first need to solve JSF's Redirect problem, discussed in the next section (*The Redirect problem with JSF*).

Before I turned to PrettyFaces, I wrote my own ViewHandler (for Facelets), which also worked with value bindings in the view mappings in the faces config file. Dominik wrote a nice article on dynamic JSF navigation, which I adapted for this approach. In the faces-config.xml I defined the target urls with the parameters added and the values then being value bindings. A navigation case would look something like the following:

```
    <navigation-rule>
     <from-view-id>/pages/newsitems.xhtml</from-view-id>
     <navigation-case>
      <from-action>#{newsitemController.doSelectNewsitem}</from-
action>
      <from-outcome>ok</from-outcome>
      <to-view-id>/pages/newsitems.xhtml?newsitem=${newsitemController
.currentNewsitem.id}</to-view-id>
      <redirect />
     </navigation-case>
    </navigation-rule>
```

The viewhandler I wrote then detected the question mark '?' in there and resolved all parameter values. I made sure the GET parameters were worked into the resulting url itself, which my PhaseListener from the previous section would be able to decipher. The result would look something like '/pages/newsitems/newsitem/4.xhtml'. This format then wouldn't be lost when going through the servlet and its filters and the PhaseListener solution presented in the previous section would understand this format, extract the parameters and fix the url for further processing by JSF, as it originally would have.

- Custom GET url generating Facelets ViewHandler with embedded parameters

2. The Redirect problem with JSF

By default JSF runs one URL behind in the browser's address bar. Every time you perform an action the JSF browser gets the URL from the previous view. This is because how JSF works with form submits:

1. You click on a button which navigates to a different page
2. JSF submits a request to itself, the same page you were on
3. JSF handles the various phases and ultimately resolves to a view id, a new page URL
4. Since JSF submitted to the same page, for the browser it will return to the same URL, while JSF renders the HTML for the next URL. This is where JSF starts to run 1 URL behind

5. The new form in the page will submit to the for JSF-current URL, which is 1 URL ahead for the browser adresbar
6. When you now click on a button, JSF submits to the JSF-current URL, at which time the browser finally catches up. JSF however simply jumps ahead again and the whole thing repeats indefinitely

The Redirect solution

To get the browser to always reflect the current JSF-viewid-resolved-URL, the servlet needs to make the browser redirect to it. If you don't, JSF will simply continue under the same URL the browser is in until the HTML form within the page submits to the next URL.

JSF actually contains something useful for this particular problem: redirects. You can add a entry in your navigation case to make the browser redirect to the resolved view URL. This pattern is known as the POST-REDIRECT-GET pattern. The small problem with this pattern however is that you'll need to add it to each and every navigation case. The big problem is that you lose your request-scoped data between the POST submit and the GET browser redirect.

The solution to both is to register a JSF PhaseListener which intercepts any response-render action, invokes a browser GET redirect instead, but saves any state information to the session so it will still be there when the browsers gets back with the GET request on the new URL. One such implementation is the PRG Phase Listener (Here is another PRG implementation by BalusC). As you can read in the comments though, there still is a shortcoming: it currently only stores the JSF messages data. Adding other data will require an arbitrary selection process since it is unclear what data will be in there to begin with (JSF internal state information stuff) and even less so what data should or shouldn't be overwritten in the session. This selection process can be cumbersome; for example, what data will you need for a richfaces datatable to still work? Will that still work when it's updated? I think you get the idea.

How PrettyFaces solves everything at once

So what PrettyFaces does is basically the culmination of the solutions I demonstrated (the GET url generation with dynamic navigation part), but polished as opposed to my own rough proof of concept. Indeed with an extra xml file making the approach generic and with a couple of other nifty features added. Telling you in one sentence what PrettyFaces does is like giving a summary of this entire post:

PrettyFaces

> *A framework for dealing with GET url in the POST based framework JSF. By returning a viewid like "pretty:home", PrettyFaces will come into action and look up how the url should be formed using the 'pretty config' xml file. It will solve any valuebindings and generate a GET url with a pattern it understand and when entering the url back into the browser, it will return the bean values from the url (sort of reverse valuebinding). So all the problems and their solutions we just discussed are solved by this small library (16k jar as of PF-1.0.0). It will redirect properly and only for the kind of queries we need GET urls for we define a pretty navigation case and return a 'pretty:viewid' string to get PrettyFaces to use it.*

In addition it offers some tags that, given a pretty maping-id, will generate the GET url for your hyperlinks. Furthermore, since version 1.2 pretty faces contains a mechanism to parse GET parameters using the traditional *?p1=v1&p2=v2* format.

For completeness, here's an overview of PrettyFaces' key classes:

PrettyContext
: *Loads your PrettyFaces context file and contains its settings and mappings.*

PrettyFilter
: *Monitors incoming requests and checks if a requested URL as mapped in the PrettyFaces context. If so, it will request the context, resolve any parameter bindings (inject values into the beans), store any FacesMessage and forward the request to the configured viewid.*

PrettyNavigationHandler
: *Handles navigation after JSF has processed a request. If a pretty-viewid was returned by the JSF action, the PrettyNavigationHandler will pick up on this, find the associated mapping from the pretty context, generate a GET url and redirect to it. In effect it determines which view to render and what appears in the browser address bar.*

PrettyViewHandler
: *Generates the requested action url from a request url and removes any trailing PrettyFaces directives from the url.*

PrettyConfig
: *Holds the mapping and is returned inside a PrettyContext instance.*

PrettyPhaseListener
: *Restores any FaceMessages stored between redirects, executes any actions defined in the mapping configuration for specific phases. This is after the beans have been injected with values from the GET url as per pretty-mapping. Finally leaves the request handling up to the PrettyNavigationHandler.*
