Proposal for dynamic configuration

Werner Punz
Apache Software Foundation - Apache MyFaces Project
Irian Solutions
Proposal for Dynamic Configurations in JSF

1. Introduction

1.1 Introduction
Following document is a proposal for following changes/extensions to the JSF API with a target version of 2.1

1. A new api which defines get, add, update and delete on every supported JSF artifact
2. New system events which allow to delegate those requests to the implementation and to additional frameworks which hook themselves into JSF

1.2 Problem
JSF currently has only static access to the configuration, which means the configuration is read by the implementation and after that no changes on configuration level is allowed.

Now this has caused problems in the past. A configuration change meant that the entire web application had to be reloaded.

Mojarra and MyFaces introduced implementation specifics which allowed if the faces-config.xml was "touched" a reload of the entire faces-config.xml based configuration, by dumping the existing configuration artifacts from ram and then reloading them.

This approach is very course grained. The aim of this document is to propose a finer grained approach without going too much into implementation details, yet.

1.3 Issues with the Status Quo
3. A full reload of the configuration in many cases means an application reset, which is especially problematic for applications which try to cover everything via JSF mechanisms.
4. Many object to object relationships in the typical JSF application only cover a hierarchy of 2-3 Objects so reloading the entire configuration just for replacing or altering a handful of artifacts is unnecessary and in many if not almost all cases overkill.
5. Since Java itself does not cover class hotplugging many solutions have arisen which try to tackle the problem of not having to restart an application or an application server by different means, yet all of them need framework hooks one way or the other to work properly. (See the Appendix A for a list of hotplugging solutions)

1.4 Common Intersection
If we look at the schematics of the average hotplugging implementation we can see common intersection points which are inherent to all solutions no matter if they are agent based (JRebel, JBoss compiler) or framework based (Ext-Scripting, Mojarra Scripting, JBoss Debug etc...)
The intersection of all approaches is a system which interferes on class loading level and on JSF configuration level.
Now, class loading has been covered by the class loader system since day 0 of Java and also some framework implementation specifics deal with it on JSF level.
The scope of this proposal is not the classloading level which is covered enough by existing implementations and specifications.

However, the access to the configuration of the JSF part currently is covered by some APIs like the Application class which allows to register some artifacts.
Let's have as example a quick look at our Application Class.
Following methods can be used for reloading the configuration

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Affected Configuration Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>addELResolver</td>
<td>ELResolver</td>
</tr>
<tr>
<td></td>
<td>setViewHandler</td>
<td>ViewHandler</td>
</tr>
<tr>
<td></td>
<td>createComponent*</td>
<td>Component</td>
</tr>
<tr>
<td></td>
<td>createConverter*</td>
<td>Converter</td>
</tr>
<tr>
<td></td>
<td>createValidator*</td>
<td>Validator</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Now Methods marked with an * are not directly defined as being able to change the configuration but can be used to simulate a configuration change.

Now as we can see the direct access to the configuration is very limited and even the simulation over the official apis only can support a subset of all needed operations and enforces proxying on the extension point side.
This problem is inherently caused by the current state of the JSF APIs and cannot entirely be resolved currently without reverting to parts of the implementation.

1.5 Configuration Change Usecases
To implement a dynamic configuration change, we can determine following coarse grained usecases:

1. Get a configuration entry
2. Add a configuration entry
3. Change a configuration entry
4. Delete a configuration entry

Every hot pluggers no matter which system has to implement those usecases, due to the fact that you only can keep the system in a consistent state if you can pass new configuration data into the running server, from the metadata acquired from the changes.

To make matters even more difficult we, since JSF 2.0, have the configuration in two locations, in the faces-config files and in the class files, which both have to be actively monitored and upon having a hot plug case, the configuration, has to be dynamically altered in the running system depending on what could be determined in both locations.

(Note the extended usecase drop configuration from ram, is implementation specific and is not part of the proposal)

1.6 Conclusion
The conclusion is simple, to get a better handling on dynamic changes in a running JSF system we need pluggable configuration access!

2 Solution

2.1 Structural Changes and Problems
The goal of the proposed structural changes is to ease the live of the developers on both sides (hot plug system authors and users of the framework), but also to keep the impact as minimal as possible.

Following usage scenarios would be possible

Extend our existing interfaces to allow the before mentioned usecases
Add a configuration singleton to allow configuration changes at a deeper level.

2.2 Proposal for a Configuration Singleton
What JSF 2 should provide is simply a configuration singleton, which allows the 4 main usecases (get, add, change, delete) for every JSF artifact.

Possible solution. FacesContext.getApplication().getConfiguration() returns a configuration singleton.
For every single JSF Artifact **get, add, change, delete** must be provided

However it is up to the implementation or the respective hot pluggers to provide the implementations on how the server has to react.
In the worst case just a a full configuration refresh is performed as we do now in the implementations by „touch faces-config.xml“.

In the best case you can get instant updates on the configuration.
In any way the end result of the request must be that a get on the configuration must reproduce the result of the last write operation and a dispatched configuration system event.

How far the amount of the implementation should go is up for discussion.
However every implementation **must** implements its behavior over the event interface described later (which probably will be the refresh mechanism we have currently in a first incarnation).

And the implementation of this behavior must be overridable, by extension frameworks which might plug themselves in.
2.4 Configuration events

2.4.1 General Problem

The specification itself should limit itself just to the general interface and distribution and
one configuration change - base mechanism implemented over the provided extension
points which just gives a common behavior (which probably will be either do nothing or
just a configuration reset)

Additionally external frameworks (Spring/CDI) currently deliver their artifacts over the
existing extension points and sometimes also keep track of the lifecycle of the artifacts
separately. So they need to be aware of requested configuration changes, to be able to
act accordingly.

2.4.2 Solution

For that kind of reason, an event based notification system should be added which
frameworks can hook into.

3.Image: JSF + Pluggable Configuration

Based on the existing SystemEvents of JSF 2.0 following events are proposed which are triggered automatically by calls against our configuration singleton.

• ConfigurationGetRequest
• ConfigurationAddRequest
• ConfigurationChangeRequest
• ConfigurationDeleteRequest

Now depending on the data delivered by those events the framework listeners can react
or not within their own configuration implementations.
It still has yet to be defined what has to be sent, but probably just the configuration data which JSF itself has to provide in an abstracted manner.

The loose coupling over an event system also would make it possible in the long run to

4. **Image Configuration System Events**

upstream the events to a global event dispatching system for such application events.
5. Image: Global Configuration Event Dispatching

2.5 Structural integrity Checks

Structural integrity checks on configuration changes should be up to the implementation or the respective reacting event handler. Since there are use cases where it is perfectly legal to have a structural breach within some time period it should be handled leniently by the implementation.

Ie: The user edits the managed bean faces-config, bean definition has a managed property to bean B it is changed to the yet non-existent bean C. The user saves the current state, some external watcher might pick up the change and notifies the running system of a configuration change for Bean A.

Bean C finally is introduced, the Daemon thread notifies the JSF runtime of a configuration add operation on Bean C

After that the user hits refresh and the EL resolves everything within the scope of the existing runtime and the integrity check is performed on demand!

2.6 Threading

The configuration singleton has to be thread save, whether the attached events handlers thread save as well, is up to the respective implementation.
3 Summary

Two extensions are requested.

1. A new API which defines get, add, update and delete on every supported JSF artifact
2. New system events which allow to delegate those requests to the implementation and to additional frameworks which hook themselves into JSF

How it is implemented is up to the corresponding implementation, but it must hook itself into the event system and the default implementation must be overridable.

Get always must produce a valid result (if nothing else is performed it must produce the current configuration) otherwise it produces what the implementation does in the configuration plugging.

Generally a default behavior is not required, it is up to the implementation if they implement the hot plugging of the configuration or not (practically every implementation probably will provide a mini hotplugging as it is done now with the „touch faces-config.xml“ implementation.
4 Appendix A: Status Quo of Solutions dealing with Hotplugging

The following list tries to give an unprecise overview of the solutions which try to deal with this problem.

1. JRebel, an instrumentation weaver which tries to cover the hotplugging issue on class level, this works mostly by replacing the original code with proxies and method delegates and introspection. JavaRebel already has a 2-3 year development behind (probably of several full time developers) of framework hooks which supplement the base classloading mechanisms.

2. JBoss javaagent solution, a yet unknown solution probably working similarly to JRebel will probably be announced officially within the March 2010-April 2010 timeframe.

3. Apache MyFaces Ext-Scripting: A framework based approach which tries to „solve“ the problem from a framework point of view. It tries to detect dependencies on class level and then unplugs existing objects from the running system via implementation hooks, so that the changed can be loaded freshely and the system is always in a consistent state.

4. Seam Debug, seems to be very similar to Ext-Scripting in its basic idea of trying to detect changes in the classes/sources, but works on a less fine grained level to avoid implementation specific hooks

5. Mojarra Scripting Support: A simple class replacement mechanism which can replace single artifacts on the fly but does not have any dependency detection

6. JSP... no further explanation is needed

7. Apache Cocoon, known that it has some kind of hotplugging but the author of this document has not yet investigated how far it goes

There are probably other solutions out there, which are at the time of writing unknown to the author.