Modulus

Creating a modular Spring application

Vladimir Ritz Bossicard (vbossica@gmail.com)

If you want to develop a modular application, almost everyone will think these days of *OSGi* [http://osgi.org/] and maybe of the *SpringSource DM Server* [www.springsource.org/dmserver]. This is all nice if you can use such platform but for many web applications out there, OSGi is simply not an option.

This project experiments with some techniques to see how to transform a Spring base application to make is look like a true modular one without going the OSGi route.

A *platform* to modularize the application

A Spring application is full of beans that are instantiated at startup. Usually, in a web application, the Spring configuration files are listed in the `web.xml` deployment file directly. This is an unwelcome side effect as soon as the same code base is used for different purposes[1]. A solution can be to multiply the `web.xml` files but I find more flexible to define one *uber-configuration* for each platform and replace the default Spring `ContextLoaderListener` in the `web.xml` by the following:

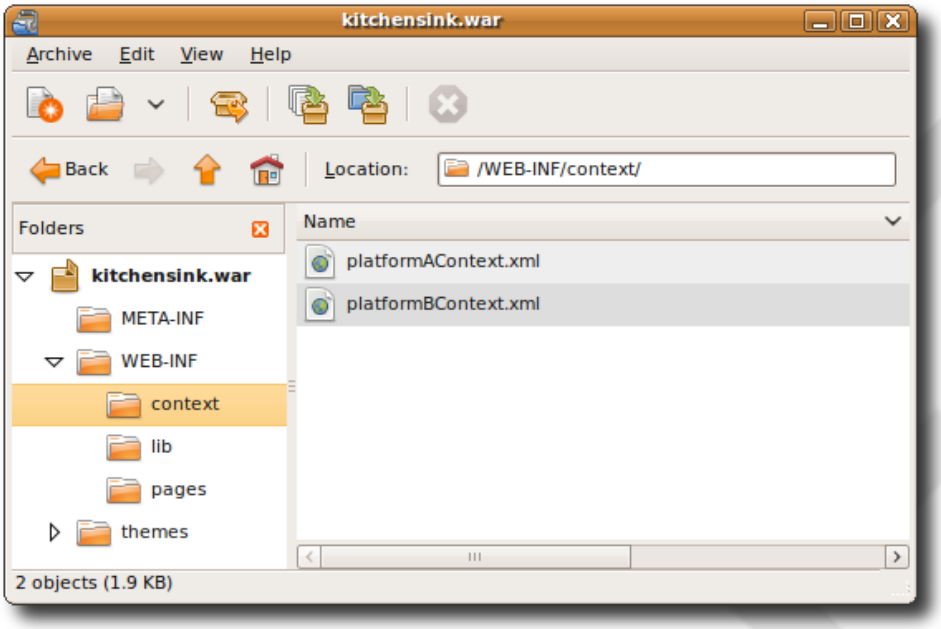
Example 1. modified web.xml configuration

```
<context-param>
    <param-name>modulus.platform.name</param-name>
    <param-value>platformA</param-value>
</context-param>

<listener>
    <listener-class>
        org.workingonit.modulus.web.PlatformContextLoaderListener
    </listener-class>
</listener>
```

Once everything has been packaged together, the application will be default start with the *platformA* but will load the *platformB* if the `-Dmodulus.platform.name=platformB` has been defined.

[1] one could start the application in a safe mode, without communication to the external systems (e.g. for testing) or used the same packaged application for different tenants.

Figure 1. Webapp packaging



Passing the first examination

The first few minutes of an application's life are not much different than the one of a newborn: everyone is examining you to check that everything is fine. To facilitate the diagnostic, *Modulus* provides several classes and interfaces to make the whole process as painless as possible.

The first step is to define a `Patient` bean in one of the application's Spring configuration files.

Example 2. Sample patient definition

```
<bean class="org.workingonit.modulus.Platform">
    <constructor-arg value="kitchensink"/>
    <property name="properties">
       <props>
           <prop key="version">5.0.12</prop>
       </props>
    </property>
</bean>
```

Some information will then automatically be logged at startup and before shutdown.

Example 3. Application startup

```
AuscultableApplicationListener INFO: Application 'kitchensink' started
        at Tue Dec 22 21:55:00 CET 2009
AuscultableApplicationListener INFO:     version: 5.0.12
```

Of course, this is just a start and far from enough to see if everything is running smoothly. The first thing to do is to define the diagnostics.

Creating simple diagnostics

Diagnostics are short, targetted and quick validations that give an overview of a (singleton) Spring bean's health. All it takes is to implement the `AuscultableBean` interface (optionally using the

`@NamedGroup` annotation) and give a `Diagnostic`. Several simple diagnostic classes are provided and it is trivial to create new one depending on the application's specific needs:

Example 4. Giving a sample diagnostic

```
public Diagnostic auscultate() {
    new DiagnosticBuilder("Service #3")
        .add(new WritableFileAuscultation("output", this.dir))
        .add(new ReacheableUrlAuscultation("Report web service", this.url))
        .build();
}
```

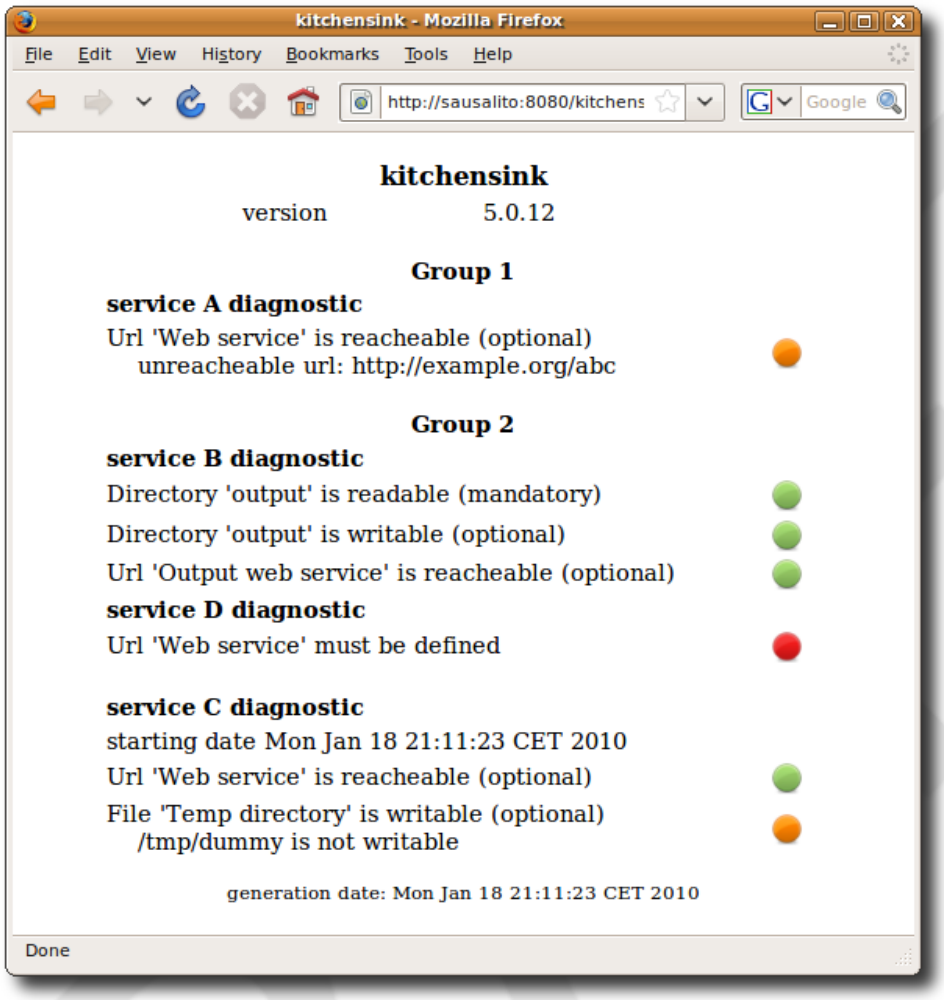
At startup, all beans implementing the `AuscultableBean` interface will be automatically registered and be made available through the (modulus) `AuscultableBeanRegistrar` bean.

Example 5. bean registrar

```
<bean class="org.workingonit.modulus.PlatformLifecycleListener"/>

<bean id="org.workingonit.modulus.AuscultableBeanRegistrar"
        class="org.workingonit.modulus.AuscultableBeanRegistrar"
    p:mbeanServer-ref="mbeanServer"
    p:prefix="com.acme.kitchensink.availability"/>
```

To diagnostics' results can either be displayed on a custom web page:

Figure 2. Sample web report



But since the `AuscultableBeanRegistrar` bean is automatically registered with the platform's JMX server, the diagnostics' results are also available to any client connecting via JMX:

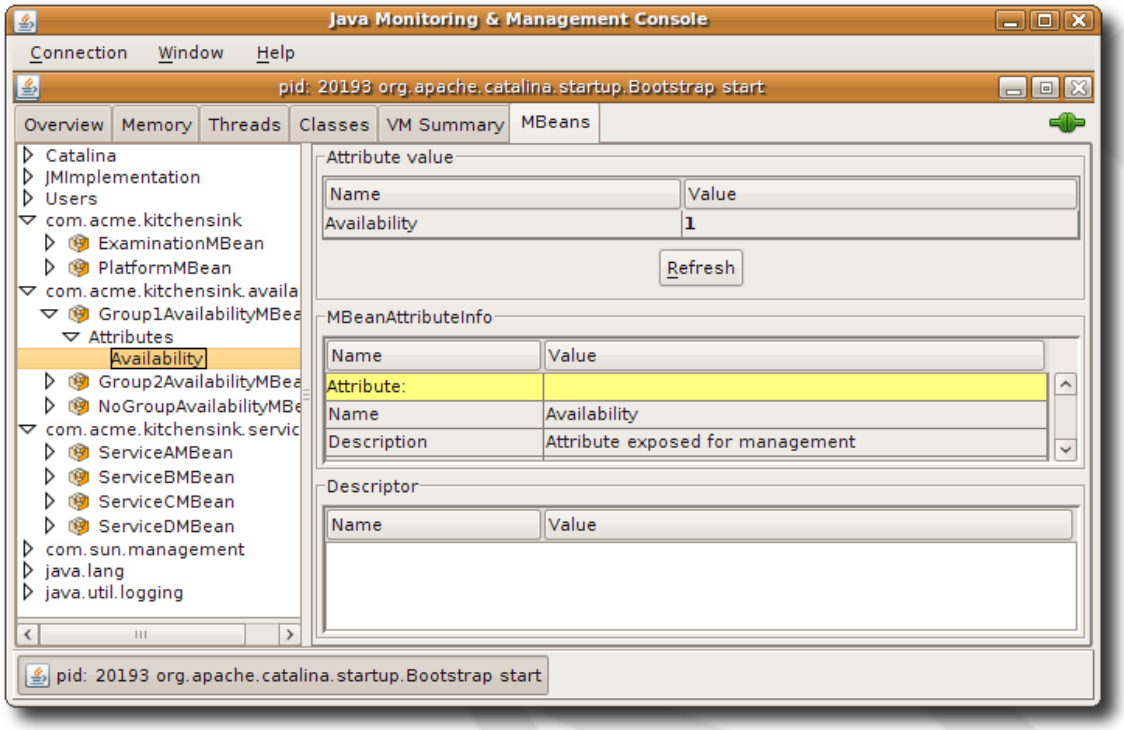
Example 6. JMX client command line

```
kitchensink
   version.....................................................5.0.12

                                  Group 1
                                  =======

service A diagnostic                                                 WARNING
   Url 'Web service' is reacheable (optional).....................WARNING
      unreacheable url: http://example.org/abc

                             ********************

                                  Group 2
                                  =======

service B diagnostic                                                    OK
   Directory 'output' is readable (mandatory).........................OK
   Directory 'output' is writable (optional)..........................OK
   Url 'Output web service' is reacheable (optional)..................OK

service D diagnostic                                                 ERROR
   Url 'Web service' must be defined..............................ERROR

                             ********************

service C diagnostic                                                 WARNING
   starting date Mon Jan 18 21:06:35 CET 2010.........................
   Url 'Web service' is reacheable (optional).........................OK
   File 'Temp directory' is writable (optional).....................WARNING
      /tmp/dummy is not writable
```

Monitoring the application

Now that the application has been installed and has passed the first checks, it starts his life in production. And the question that then needs to be continuously answered in is if the application still runs. Since most monitoring servers support JMX, it is very easy to reuse the diagnostics created previously and expose them via JMX. And that's exactly what *Modulus* does, you don't have to do anything more.

Figure 3. JMX exported services



What left to do is to use the *Modulus* JMX client and generate the configuration file for the Hyperic monitoring server.

Figure 4. Application monitored by Hyperic