Clusterus

Webapp meet cluster, cluster meet webapp

Vladimir Ritz Bossicard (vbossica@gmail.com)

After spending time in Kindergarden, your application is now ready to play with the big guys and be installed in a cluster. Load balancing and failover are now within reach! This project plays with a couple of experiments that can be very useful when deploying a spring based application into a cluster.

Session Creation

Usually you develop a web application, but normally it's more a server application with a web front end. This means that the application can be invoked without the web (via web services or JMS messages). Having a spring application, it is normal to look at the spring portfolio for the security aspect. The following example is based on the spring-security framework but the design is general enough that it can be modified to fit other frameworks as well.

`SessionContext` contains the minimal set of information to execute business functions.
The authentication and the subsequent creation of the `SessionContext` is done by the `EnhancerUserDetailsService`.

Figure 1. SessionContext definition

```
import org.springframework.security.userdetails.UserDetails;

public class SessionContext implements UserDetails {

    private Map<String, Object> clustered = new HashMap<String, Object>();
    private transient Map<String, Object> value = new HashMap<String, Object>();

    public void addClusteredValue(String key, Object value) {
        this.clustered.put(key, value);
    }

    public void addTransientValue(String key, Object value) {
        this.value.put(key, value);
    }

}
```

The Spring-security framework has been primarily developed for a web environment but it is not too difficult to create a custom class to provide a headless authentication service. That's what the `HeadlessAuthenticationProvider` is all about. After that, gluing everything together is just a matter of defining a couple of spring beans.

Figure 2. SessionContext initialization

```
import org.springframework.security.userdetails.UserDetails;

public class SessionContext implements UserDetails {

    private transient Map<String, Object> value = new HashMap<String, Object>();

    public boolean isInitialized() {
        if (this.value == null) {
            this.value = new HashMap<String, Object>();
        }
        return !this.value.isEmpty();
    }

}
```

Figure 3. Spring security configuration

```
<bean id="authenticationManager"
        class="org.springframework.security.providers.ProviderManager">
    <property name="providers">
        <list>
            <ref bean="authenticationProvider"/>
        </list>
    </property>
</bean>

<bean id="headlessAuthenticationProvider"
        class="org.workingonit.clusterus.security.HeadlessAuthenticationProvider">
    <property name="authenticationProvider" ref="authenticationProvider"/>
</bean>

<bean id="authenticationProvider"
        class="org.springframework.security.providers.dao.DaoAuthenticationProvider">
    <security:custom-authentication-provider/>
    <property name="userDetailsService" ref="userDetailsService"/>
</bean>

<bean id="userDetailsService"
        class="org.workingonit.clusterus.security.EnhancerUserDetailsService">
    <property name="pathThrough">
        <!-- original UserDetailsService -->
    </property>
</bean>
```

HTTP Session Validation

When the term *cluster* is heard, the first this that pops up is usally *HTTP session serialization*. In order to fullfill the high availability of a cluster, the elements of the HTTP session must be copied to the other nodes of the cluster, so that users are automatically "migrated" to a new server when one goes down. In order to keep things run smoothly, one should make sure that the objects in the HTTP session:

- are *serializable*: in Java, the classes should simply implement the `Serializable` interface (although there are other tricks when they don't, but let's not go there).
- are *not too big*: the network will definitively not survive if it has to replicate megabytes amongst the cluster's members.

The particularity of these requirement is that they must be verified at *runtime*. A simple servlet filter would do it. Please meet `HttpSessionIntrospectorFilter`.

Once per request, all the elements of the HTTP session will be individually validated for serializability. Every non serializable object found in the HTTP session will be copiously mentioned in the application log:

Figure 4. Non serialization session

```
WARNING: session object not serializable: SPRING_SECURITY_CONTEXT
java.io.NotSerializableException:
    com.acme.kitchensink.web.session.DefaultSessionContextInitializer$NonSerializable
at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1156)
at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:326)
```

The total size of these objects will also be calculated and logged, as a frenge benefit:

Figure 5. Session size information

```
INFO: Estimated session size : 1570 [A10BE5E41B17581689668509D26A5D3F]
[login]
INFO: Estimated session size : 2866 [77FCD90AB01A6340BDADA10C18292313]
INFO: Estimated session size : 3427 [77FCD90AB01A6340BDADA10C18292313]
INFO: Estimated session size : 3685 [77FCD90AB01A6340BDADA10C18292313]
[logout]
INFO: Estimated session size : 1659 [0CE893E7FE4437363C6C6DEBD1544AC3]
```

By parsing the log files, it is quite easy to check that the size stays between tolerable borders. Of course, the extra mile would be to graph the size in real time but that exercise if left out for later.

Business Singleton

One of the advantages of deploying an application into a cluster is of course to benefit from its fail over capabilities: as soon as an application goes down (voluntarily or not), another one steps up to the plate to continue the work. The only restriction is that all the applications must be active at any given time. I voluntarily simplify a situation that can be complicated at will with a cloud infrastructure.

Anyway, it is sometimes desired to only have one single application (I call this beast a *business singleton* but you're free to coin another term) running in the cluster at any given time but without renouncing to the fail over capabilities if the server hosting that singleton suddenly have some technical difficulties.

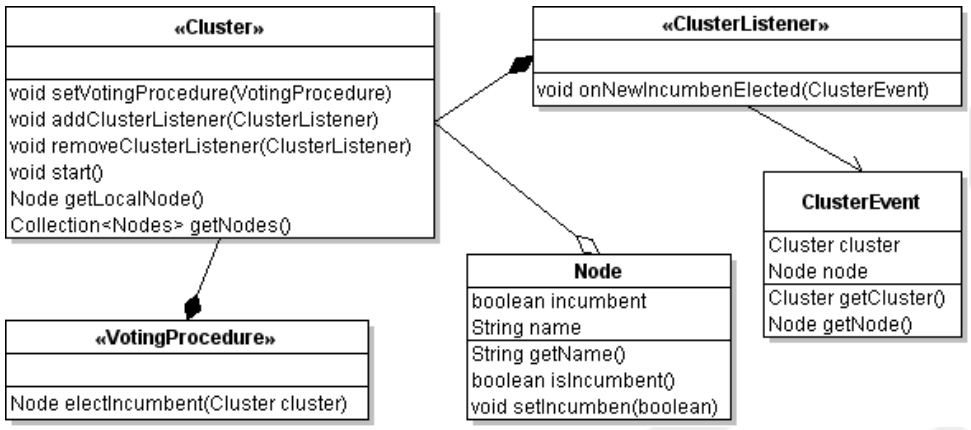
The starting idea of this project is to provide a framework that enables an application to be installed on each node but ensure that only one instance is active at any given time and that another instance will automatically be activated should the previous one fail.

The two aspect of the solution revolve around the notions of a democratic cluster and a singleton dictator.

The Democratic Cluster

The *democratic* cluster is a cluster that is able to automatically elect one of its members to be the übernode. As soon as this übernode has disappeared from the cluster, another election takes place and another node has the privilege to take over the duty. The next figure depict the classes involved into the cluster modelling:

Figure 6. Democratic cluster model



As you can see, the model is rather simple and can be explained in one sentence: when a `Node` joins a `Cluster` an election takes place on each member of the cluster and a particular voting procedure is followed to determine the incumbent; once the decision is fallen, the `ClusterListeners` are notified with the reception of the *new election* `ClusterEvent`.

For this prototype, the *JGroups library* [http://www.jgroups.org/] -an Open Source toolkit for reliable multicast communication- has been used to implement basic `Cluster`. Since the `Cluster` design is technology agnostic, other protocols or transports (like JMS) could also be used. The Spring configuration for using this implementation is shown in the next code fragment:

Figure 7. JGroups Cluster configuration

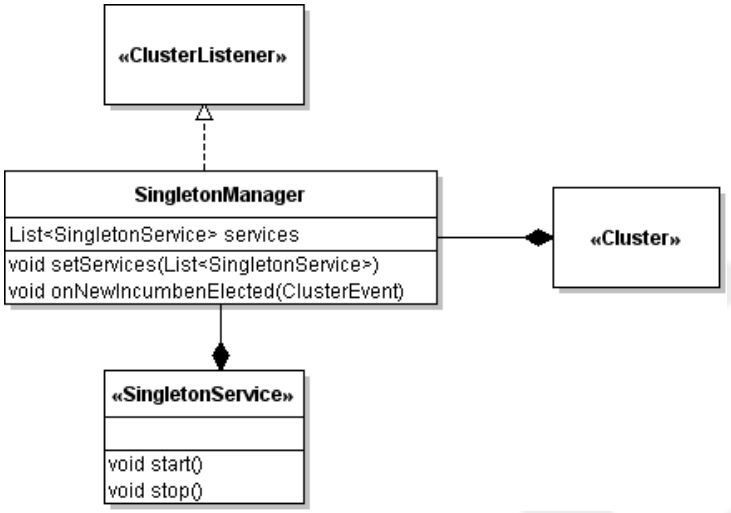
```
<bean id="cluster" class="org.workingonit.singulus.jgroups.JGroupsCluster">
    <property name="channel">
        <bean class="org.workingonit.singulus.jgroups.ChannelBeanFactory">
            <property name="config" value="classpath:jgroups-config.xml"/>
        </bean>
    </property>
    <property name="votingProcedure">
        <bean class="org.workingonit.singulus.election.IpLexicalVotingProcedure"/>
    </property>
</bean>
```

Once the incumbent `Node` has been designated, the only thing left to do is to manage the business singletons. This is the responsibility of the *singleton dictator*.

The Singleton Dictator

A *singleton dictator* is simply responsible of managing the services that are considered as *business singletons*. In our example application, the `SingletonManager` is responsible for starting and stopping several `SingletonServices` objects. The class diagram is described in the next figure:

Figure 8. SingletonManager implementation



The Spring configuration for a basic singleton manager is trivial but Since the only dependencies are against the `ClusterListener` and `Cluster` classes, other implementations could easily be developed to address more challenging scenario.

Figure 9. Acme Kitchensink Spring configuration

```
<bean id="service1" class="com.acme.kitchensink.services.SampleSingletonService"/>

<bean id="service2" class="com.acme.kitchensink.services.SampleSingletonService"/>

<bean class="org.workingonit.singulus.SingletonManager">
    <property name="cluster" ref="cluster"/>
    <property name="services">
        <list>
            <ref local="service1"/>
            <ref local="service2"/>
        </list>
    </property>
</bean>
```

Finally let's see how everything duct tails together by looking at how a cluster with two nodes would work together.

Sample Execution

Once our application has been implemented and configured, we're ready to run it in a cluster. For this example, we will take the minimalistic approach of running the application in a two nodes cluster. So, *Ladies and gentlemen, start your applications!*

As soon as the first application has been started, an election takes place and it won't be a surprise to anyone that this single node is declared the winner. As a consequence the services under its management are started.

Figure 10. Server #1 starting up

```
-------------------------------------------------------
GMS: address is 192.168.0.101:1390 (cluster=SINGULUS)
-------------------------------------------------------
org.workingonit.singulus.SingletonManager  - new incumbent elected = 192.168.0.101:1390
org.workingonit.singulus.SingletonManager  - starting services
com.acme.kitchensink.services.SampleSingletonService  - Starting service service1
com.acme.kitchensink.services.SampleSingletonService  - Starting service service2
```

The second server is now started and an election is also taking place. This time however, the incumbent has already been elected and no singleton service is initialized. Although two applications are now running concurrently, only one single instance of each *business singleton* is running.

Figure 11. Server #2 starting up

```
-------------------------------------------------------
GMS: address is 192.168.0.101:1392 (cluster=SINGULUS)
-------------------------------------------------------
org.workingonit.singulus.SingletonManager  - new incumbent elected = 192.168.0.101:1390
org.workingonit.singulus.SingletonManager  - services already stopped
```

Let's now take a look at what's happening in the first application. After the second server has joined the cluster, an election has also taken place but since the local node (`192.168.0.101:1390`) was already the reigning incumbent, no service was started.

Figure 12. Service #1 seeing #2 showing up

```
-------------------------------------------------------
GMS: address is 192.168.0.101:1390 (cluster=SINGULUS)
-------------------------------------------------------
org.workingonit.singulus.SingletonManager  - new incumbent elected = 192.168.0.101:1390
org.workingonit.singulus.SingletonManager  - starting services
com.acme.kitchensink.services.SampleSingletonService  - Starting service service1
com.acme.kitchensink.services.SampleSingletonService  - Starting service service2
org.workingonit.singulus.SingletonManager  - new incumbent elected = 192.168.0.101:1390
org.workingonit.singulus.SingletonManager  - services already started
```

The next and final step is to make sure that the cluster can survive if the incumbent is `kill -9`'ed by an unscrupulous system administrator or overwhelmed by a bubbling `NullPointerException`.

We can see that the server #2 has received the notification that a node a disappeared from the cluster and a new election takes place. Since the incumbent place is now vacant, it's another node that is elected, according to the defined `VotingProcedure` and it's again no surprise that the last application standing, in our case the server #2, is chosen and promptly starts its registered `SingletonService`.

Figure 13. Server #2 taking the hand

```
-------------------------------------------------------
GMS: address is 192.168.0.101:1392 (cluster=SINGULUS)
-------------------------------------------------------
org.workingonit.singulus.SingletonManager  - new incumbent elected = 192.168.0.101:1390
org.workingonit.singulus.SingletonManager  - services already stopped
org.workingonit.singulus.jgroups.JGroupsCluster  - suspected address: 192.168.0.101:1390
org.workingonit.singulus.SingletonManager  - new incumbent elected = 192.168.0.101:1392
org.workingonit.singulus.SingletonManager  - starting services
com.acme.kitchensink.services.SampleSingletonService  - Starting service service1
com.acme.kitchensink.services.SampleSingletonService  - Starting service service2
```

That's it, we're done! We can now define a cluster with an unlimited number of nodes and still make sure that only one instance of each *business singleton* will be running at any given time. Furthermore, everything is automatic and is applicable from standalone to web and Java EE applications.