

Working with Mule and Newton: Getting Started

Introduction

Mule is a very popular open source ESB and application toolkit. It has many features which are popular with it's users including: ease of configuration, a rich eco-system of transport adapters and out of the box components. However it is not fundamentally designed to run on a multi-node distributed environment, relying upon a container to provide services that relate to stability and scalability. In this series of articles we'll be looking at how you can design enterprise grade systems with Mule and it's open-source cousin Newton.

Newton is an OSGi based distributed application server which is designed from the bottom up for extreme scalability and stability. Eschewing the traditional master-server clustering approach for a rich genetic based network - where any node can take on any role - it aims at 100% stability rather than the traditional good-enough approach. Newton is an open-source product released under the AGPL license but also has a big brother in the guise of Infiniflow; it's commercial counterpart sold by Paremus, a UK based company.

Prerequisites

You'll need a good working knowledge of Maven to make the most out of the article; it is assumed that you are already well acquainted with Mule but are unfamiliar with Newton. It would be helpful to have a brief understanding of OSGi - you may wish to take a look at a tutorial such as this: http://www.knopflerfish.org/tutorials/osgi_tutorial.pdf; and it may help to try out the demos on the Newton website <http://newton.codecauldron.org/Demos>.

Before we start

Downloading Newton

Newton can be downloaded from <http://newton.codecauldron.org/Download+and+Install> . Download and unzip/untar the distribution.

Building Newton

Building Newton from source is straightforward, just go into the top level directory of the distribution and type:

```
ant -lib global-build/lib/ant-lib
```

This will build Newton; the built installation can be found in **build/install** and the executables in **build/install/bin**.

Getting Started

In this first article we're going to focus purely on running Mule on Newton rather than in depth integration between the two applications.

There is no practical reason why you can't take your existing project and deploy right onto Newton with only a minimum of work. However there are a few new concepts that you need to be familiar with before launching head on, so let's start with the simplest approach and use a pre-built project.

Maven supports a useful feature for producing ready made projects from templates; these are archetypes. Fortunately for you, there is an archetype available for producing a skeleton of a Mule and Newton based project. This is part of the Mule4Newton project here: <http://mule4newton.codecauldron.org/>.

So the general command is:

```

mvn archetype:create -DarchetypeArtifactId=newton-mule-archetype \
-DarchetypeGroupId=org.cauldron.newton.archetype \
-DartifactId=<project-name> -DgroupId=<project-package-name> \
-DarchetypeVersion=1.0-SNAPSHOT \
-Dversion=<project-version> \
-DremoteRepositories=http://mvn.codecauldron.org/artifactory/repo/

```

Please make sure that you specify a purely numeric value for **<project-version>** as this will be used to identify the version of the OSGi bundle produced.

Creating a Sample Project

For the purposes of our little tutorial let's start by executing this command.

```

mvn archetype:create -DarchetypeArtifactId=newton-mule-archetype \
-DarchetypeGroupId=org.cauldron.newton.archetype -DartifactId=mule4newton \
-DgroupId=org.testing -DarchetypeVersion=1.0-SNAPSHOT -Dversion=0.1 \
-DremoteRepositories=http://mvn.codecauldron.org/artifactory/repo/

```

This will create a new directory called mule4newton.

```
cd mule4newton
```

In this directory you will find a Maven POM file called **pom.xml** and a source tree. The POM contains the basics for producing Newton compatible OSGi bundles and the dependencies for an example Mule project. The source tree contains the code for a (very) simple Mule project which can be deployed to Newton. You may wish at this point to create a project structure for IntelliJ or Eclipse. IntelliJ 7 supports Maven projects natively so need to do this if you use IntelliJ or are just happy hacking around without an IDE.

```

mvn idea:idea
mvn eclipse:eclipse

```

If you are familiar with Maven you will probably have noticed the packaging type is bundle:

```
<packaging>bundle</packaging>
```

The plugin which is used to produce the bundle is specified later on in the configuration here:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <version>1.2.1-SNAPSHOT</version>
      <extensions>true</extensions>
      <configuration>
        <instructions>
          <Bundle-Version>0.1</Bundle-Version>
          <Private-Package>*; -noimport:=true</Private-Package>
          <Import-
Package>org.cauldron.newton.runtime.activator;specification-
version=1.0,org.cauldron.newton.component.context,org.cauldron.newton.framework;specifica
tion-version=1.0,!*</Import-Package>
          <Bundle-Category>services</Bundle-Category>
          <Bundle-
Activator>org.cauldron.newton.installer.component.client.Activator</Bundle-Activator>
          <Installable-Component>true</Installable-Component>
          <Installable-Component-Templates>mule-template.xml</
Installable-Component-Templates>
          <_nouses>true</_nouses>
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>

```

```

        </configuration>
    </plugin>
</plugins>
</build>

```

Breaking this down a little:

```

    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <version>1.2.1-SNAPSHOT</version>

```

The plugin used to generate the bundle is supplied by the Felix project, Felix is an open-source OSGi container. The version is 1.2.1-SNAPSHOT so is possibly unstable - life on the bleeding edge! We now configure the plugin for our project in the instructions section:

```
<Bundle-Version>0.1</Bundle-Version>
```

So this is the version to use for the bundle and is the version number we gave the archetype for the project.

```
<Private-Package>*; -noimport:=true</Private-Package>
```

Next we specified that we're not intending to export or import any of the versions of the classes used in this bundle - we're going to make them all private. This a very non-OSGi approach which we've taken since Mule is not (yet) configured as OSGi bundles. So now we import the few packages that we know are provided as OSGi bundles by the system and - as mentioned above - exclude all others from being imported.

```
<Import-Package>org.cauldron.newton.runtime.activator;specification-
version=1.0,org.cauldron.newton.component.context,org.cauldron.newton.framework;specifica
tion-version=1.0,!*</Import-Package>
```

This line is standard for any server-side OSGi bundle.

```
<Bundle-Category>services</Bundle-Category>
```

Next we need to provide a standard activator for the OSGi bundle - an activator is a lifecycle callback in OSGi, here we're using the activator supplied by Newton:

```
<Bundle-Activator>org.cauldron.newton.installer.component.client.Activator</Bundle-
Activator>
```

Now we're going to provide the information required by Newton itself:

```
<Installable-Component>>true</Installable-Component>
<Installable-Component-Templates>mule-template.xml</Installable-Component-Templates>
```

This tells Newton that this is not just a vanilla OSGi bundle, but actually contains an SCA component and to use the **mule-template.xml** file to describe the component. This template forms the basis of any deployed instance of the service, individual deployed instances can override settings specified in the template.

So there we have it then, that's are bundle configured, we just need to specify where to get the Felix plugin from. In this case we're going to use the Apache Maven 2 Snapshot repository:

```

<repositories>
  <repository>
    <id>apache.m2.incubator</id>
    <name>Apache M2 Incubator Repository</name>
    <url>http://people.apache.org/repo/m2-snapshot-repository</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>

```

```

        <id>apache.m2.incubator</id>
        <name>Apache M2 Incubator Repository</name>
        <url>http://people.apache.org/repo/m2-snapshot-repository</url>
    </pluginRepository>
</pluginRepositories>

```

Understanding the Sample Project

The sample project is an extremely trivial Mule project, the configuration for Mule can be found in **src/main/resources/mule-config.xml** and looks like this:

```

<!DOCTYPE mule-configuration PUBLIC "-//MuleSource //DTD mule-configuration XML V1.0//EN"
    "http://mule.mulesource.org/dtds/mule-configuration.dtd">

<mule-configuration id="Configuration" version="1.0">
...

    <model name="Model">

        <mule-descriptor name="Emitter"
            implementation="{groupId}.Emitter">
            <inbound-router>
                <endpoint address="quartz:/myService">
                    <properties>
                        <property name="repeatInterval" value="1000" />
                        <property name="payloadClassName" value="java.lang.Object" />
                    </properties>
                </endpoint>

            </inbound-router>
            <outbound-router>
                <router className="org.mule.routing.outbound.OutboundPassThroughRouter">
                    <endpoint address="stream://System.out"/>
                </router>
            </outbound-router>
        </mule-descriptor>
    </model>

</mule-configuration>

```

This configuration uses a quartz timer to trigger the execution of the Emitter service every second which then emits the time in milliseconds. The outbound router routes this to **System.out**. So when deployed we see the time in milliseconds being displayed once a second. For completeness here is the **Emitter** class:

```

public class Emitter
{

    public String emit(Object o) {
        return System.currentTimeMillis()+"\n";
    }
}

```

The glue between Newton and Mule is the **MuleAdapter** class, this you will find in the same package as the Emitter class. For many projects you will find the **MuleAdapter** is all that you need; however for some you may wish to customize it. So let's take a moment to understand the class first.

```

public class MuleAdapter implements ComponentLifecycleListener

```

The first thing you will notice is that it implements **ComponentLifecycleListener**. This is a Newton interface that allows deployed services to be aware of the lifecycle of the component they are part of. This interface requires three methods to be implemented but the two we're interested in are **initialise()** and **destroy()**. In the implementation of the **initialise()** method the class first checks to see if a mule instance is already configured

(purely as a safety check) and then if it hasn't it configures a new instance based on the **configurationFile** property.

```
public void initialise(ComponentContext componentContext) {
    try
    {
        if(manager == null) {
            MuleXmlConfigurationBuilder builder = new MuleXmlConfigurationBuilder();
            manager = builder.configure(configurationFile);
        } else {
            throw new RuntimeException("Initialized twice.");
        }
    }
    catch (ConfigurationException e)
    {
        throw new RuntimeException(e);
    }
}
```

The **configurationFile** property can be the name of a single configuration file or a comma separated list. To make sure Mule is shutdown correctly when the service is undeployed the **destroy()** method is implemented also.

```
public void destroy() {
    try
    {
        manager.stop();
        manager= null;
    }
    catch (UMOException e)
    {
        throw new RuntimeException(e);
    }
}
```

Building our Sample Project

Before we can build the project we need to tell Maven where to go for Newton dependencies. At the time of writing Newton is not built using Maven, but as a convenience Newton artifacts are provided by the Code Cauldron repository. So we need to add this to our Maven settings

```
<repositories>
  <repository>
    <id>code-cauldron</id>
    <url>http://mvn.codecauldron.org:80/artifactory/repo/</url>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>code-cauldron-snapshots</id>
    <url>http://mvn.codecauldron.org:80/artifactory/repo/</url>
    <releases>
      <enabled>>false</enabled>
    </releases>
  </repository>
</repositories>
```

The snapshot repository entry is not essential for this project but why not add it at the same time.

Okay, now we've added that to our **settings.xml** we're ready to rock'n'roll, so inside our **mule4newton** directory we build our project:

mvn install

Inside the newly created target directory you will find a **mule4newton-0.1.jar** (which will now also be in your local Maven repository).

Deploying a Single instance to Newton

So inside the build/install/bin directory of Newton we have a **container** (or **container.bat** on Windows) executable. Let's run that

```
./container
```

Great, now we have Newton running. All we have to do is deploy our bundle and then install an instance of the composite. So first things first let's install the bundle on the Newton command line type:

```
cds scan boot <path-to-mule4newton>/mule4newton/target
```

The **cds scan** command looks for bundles to deploy - we specify **boot** as we're only interested in, at this early stage, deploying to a single Newton node so we deploy to the boot zone on the local instance only. When we cluster this later on we'll specify **remote** instead. Next we install an instance of the bundle:

```
installer install <path-to-mule4newton>/mule4newton/src/main/resources/  
instance.xml
```

This uses the provided **instance.xml** to specify the specifics of the instance to be deployed, here is the **instance.xml** produced by the archetype:

```
<?xml version="1.0"?>  
<composite name="org.testing.Mule4NewtonExampleInstance">  
  <bundle.root bundle="org.testing.mule4newton" version="0.1"/>  
  <include name="org.testing.Mule4NewtonTemplate"/>  
  <property name="configurationFile" value="mule-config.xml" type="string" />  
</composite>
```

It specifies the bundle containing the resources required for the composite instance and the template on which the instance is based. It also contains a property which specifies where to find the mule configuration file.

After a short delay you should now see a steady stream of numbers appearing indicating the successful deployment.

Deploying Multiple Instances to Newton

Now here's the interesting bit, consider all our activities up until this point a prelude. The reason why we're running Mule on Newton is because Newton is designed for distributing applications in a self-healing adaptive network and this gives us failover and scale-out. In the next article we'll also be looking at how Mule and Newton can interact more to further improve these characteristics.

So let's get going, this time we're going to start more than one container and we'll tell it which fabric to be part of and which numeric node it is.

```
./container -fabricName=fabric -instance=1
```

and then our second node in another window:

```
./container -fabricName=fabric -instance=2
```

After waiting patiently for a little while the command line will appear and in each window we type:

```
system manage etc/systems/remote-container.system
```

In the first window we're going to start up the command line for running as a manager so we type:

```
exec etc/scripts/peer-node
exec etc/scripts/management-node
exec etc/scripts/cds-registry
exec etc/scripts/system-manager
exec etc/scripts/cli-tools
```

Now to deploy our bundle from the project:

```
cds scan remote /Path/To/mule4newton/target
```

And then we use the supplied system.xml file:

```
remote-system manage /Path/To//mule4newton/src/main/resources/system.xml
```

This will deploy our Mule based project to both nodes. Try running another container with:

```
./container -fabricName=fabric -instance=3
```

Nothing happens! Now go and stop the first container, by any means necessary, wait for a little while and low and behold, our application is running on the new node.

Understanding the System Description File

The **system.xml** file is used to describe a multi-node system in Newton and depending on the configuration supplied will decide how the distributed system will change according to node failure, increase in load etc.

```
<system name="mule4newton-system">
  <description>An example static system</description>

  <system.composite name="mule4newton-system-composite-1"
    bundle="org.testing.mule4newton"
    template="org.testing.Mule4NewtonTemplate"
    version="0.1">
    <replicator name="scale-1-to-2"/>
    <property name="configurationFile" value="mule-config.xml" type="string"/>
  </system.composite>

  <replication.handler name="two"
    type="org.cauldron.newton.system.replication.FixedSizeReplicationHandler">
    <property name="size" value="2" type="integer"/>
  </replication.handler>

  <replication.handler name="three"
    type="org.cauldron.newton.system.replication.">
    <property name="size" value="3" type="integer"/>
  </replication.handler>

  <replication.handler name="scale-1-to-2"
    type="org.cauldron.newton.system.replication.ScalableReplicationHandler">
    <property name="scaleFactor" value="1" type="float"/>
    <property name="minimum" value="1" type="integer"/>
    <property name="maximum" value="2" type="integer"/>
  </replication.handler>

</system>
```

In our extremely simple example we describe the system composite as being based on the template **org.testing.Mule4NewtonExampleInstance** which can be found in **mule-template.xml**. We then specify that it should be replicated using the replication handler called “**scale-1-to-2**”. This replication handler tells Newton to maintain between one and two instances of our system composite across the fabric at any given time. If you would like to know a little more about how systems are managed and provisioned then take a look here: <http://newton.codecauldron.org/System+Managers%2C+Provisioners+and+Containers>.

In Conclusion

So we've taken a brief look at how to run Mule on Newton, this means now that we now produce n-tier and other topologies for our Mule based applications just by changing a single system.xml. We can now run multiple Mule applications together on the Fabric without fear of class loading issues thanks to OSGi. We can deploy updates to multiple nodes with a single command. In future articles we'll be taking a look at how to create dynamic topologies that change with load and how to do efficient load-balanced messaging across an auto-scaling deployment environment.