Team Development with Mule

Team Development with Mule

Team Development with Mule

Topics in these books have addressed your local workstation, and a single project with a single configuration file. When working on a team, the Mule project will increase in size, will increase in its number of developers, and must run in other environments, such as test and production. Here are some practices that will make such growth possible.

The different approaches to modularizing Mule configurations and applications are all opportunities for splitting work across teams, whether these teams work on the same overarching project or on different projects with an accent put on reuse.

Modularizing Your Configuration Files for Team Development

Modularizing Your Configuration Files for Team Development

Though it may seem convenient to keep all your Mule configuration in one place, the reality is that a gigantic XML file quickly becomes unmanageable. This is why it is recommended to split monolithic configurations into several files and leverage Mule's capacity to load multiple configuration files at application start-up time. Moreover, splitting configurations into multiple fragments encourages re-use across teams.

Mule offers two options for loading several configuration files:

*side-by-side: provide a list of independent configuration files to load,
*imported: have one configuration file import several others, which in-turn can import other files.

In practice, it is common to use both approaches simultaneously.

Don't forget that all the configuration files end up loaded in the same context; therefore you should be careful and use unique names for all your configuration elements. Mule will refuse to load an application whose configuration files contain name conflicts.

How can you determine what constitutes good separation lines between configuration fragments? Here are a few rules of thumb:

*Business domains usually form a natural border that can be used to separated configuration elements
*Keeping together elements that have similar reasons for change reduces the risk of impacting unrelated aspects of your application
*Technical aspects, like administrative components, security or Spring beans configuration, define good lines of demarcation
*Extracting a side-by-side transport configuration (connectors and endpoints) facilitates functional testing (discussed in section 3). Note that it is not intended to take care of environment specific transport configuration, which is dealt with properties files (discussed later on)
*And, last but not least, re-use across teams and projects (also discussed later on)

Imported configuration files

Mule relies on Spring XML configuration for importing configuration files into each other.

Here is the main configuration file illustrated above, which takes care of importing the three other configuration elements:

```
<mule xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:spring="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="
      http://www.mulesoft.org/schema/mule/core
             http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd
      http://www.springframework.org/schema/beans
             http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <spring:beans>
    <spring:import resource="domain-A-config.xml" />
    <spring:import resource="domain-B-config.xml" />
    <spring:import resource="admin-config.xml" />
  </spring:beans>
</mule>
```

Using Side-by-Side Configuration Files

Using Side-by-Side Configuration Files

Side by side configurations are independent and require nothing specific to work except to let Mule know it should load them.

To do so, create a file named mule-deploy.properties in the same directory that contains your configuration files and add configuration similar to the following one, but of course with your configuration file names:

```
config.resources=mule-main-config.xml,mule-transports-config.xml
```

With this in place, Mule will know which configuration files it should load when deploying your application.

If you're starting your application from Eclipse, go to the parameters screen of the "Run" configuration you use and select your files there.

Using Parameters in Your Configuration Files

Using Parameters in Your Configuration Files

When an application gets deployed in different environments, like QA, pre-production or production, it usually needs to be configured differently as server names, credentials and other similar parameters will vary.

As a developer facing this kind of variability, your goal is to produce a single Mule application for all your environments and to externalize all the environment-specific configuration parameters. This is the key to reproducible deployments.

Consider externalizing other aspects of your configuration, like time-out values, polling frequencies, etc... even if they don't vary between environments. This will facilitate tuning and experimenting as the whole Mule application would become configurable through a single properties file.

In Mule, you achieve this by using Spring's property placeholder resolution mechanism. Consider the following Mule configuration fragment that defines an HTTP endpoint pointing to a password protected web resource:

```
<http:endpoint name="ProtectedWebResource"
               user="${web.rsc.user}"
               password="${web.rsc.password}"
               host="${web.rsc.host}"
               port="80"
               path="path/to/resource" />
```

The variables bits are clearly visible: the user, password and host can vary for each environment where this endpoint gets deployed in. To provide

values for these variables, we use a standard Java properties file:

```
web.rsc.user=alice
web.rsc.password=s3cr3t
web.rsc.host=www.acme.com
```

Use a consistent naming strategy for your properties and make them unique across applications: this will greatly facilitate re-use across teams.

You also need to configure Spring's property placeholder configurer. Instead of configuring Spring to load a single properties file, follow this approach:

- Configure Spring to load a default properties file and another file containing overrides.
- Ship a default properties file with values applicable for developers' workstations inside your Mule application deployable.
- Create the properties override file only in the environments where it's needed and with only the properties that actually need to be overridden.

The advantages of this approach are:

- Developers don't need to deploy and run the application locally.
- The ops team only needs to work with the set of properties they have to configure for a particular environment.

Here is a method of accomplishing this:

```
<mule xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
      http://www.mulesoft.org/schema/mule/core
            http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd
      http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-3.0.xsd">
  <spring:beans>
    <context:property-placeholder
             location="classpath:my-mule-app.properties,
                       classpath:my-mule-app-override.properties" />
  </spring:beans>
</mule>
```

With this in place, add a `my-mule-app.properties` file in your application resources directory (`src/main/app` for a Mule application Maven project) and put default and development environment values in it. To override some values, create a my-mule-app-override.properties file and drop it in `$MULE_HOME/conf`.

If your ops team can't drop files in Mule's directory hierarchy, the alternative is to configure the placeholder configurer to pick up the override file from a well-known location, as shown here:

```
<context:property-placeholder
         location="classpath:my-mule-app.properties,
                   file:///etc/mule/conf/my-mule-app-override.properties" />
```

Use unique file names for your properties files to ease the burden on sysadmins. A good strategy is to use the application name or ID in the default and override properties file names.
Should you need to encrypt passwords in your properties file, consider using Jasypt's subclass of Spring's placeholder configurer. For more information, check the section named "Encryption-aware Spring's property configurers" on this page:
http://www.jasypt.org/encrypting-configuration.html

Using Modules In Your Application

Using Modules In Your Application

Mule applications themselves can provide an additional way to modularize your project.

Thanks to Mule's capacity to run and hot-redeploy applications side-by-side in the same instance, such a coarse-grained approach to modularity is useful when keeping elements of your application running while others could go through some maintenance operations.

For optimum modularity:

- Consider what services are tightly interrelated and keep them together in the same Mule application: they will form sub-systems of your whole solution.
- Establish communication channels between the different Mule applications: the VM transport will not be an option here, as it can't be used across different applications. Prefer the TCP or HTTP transports for synchronous channels and JMS for asynchronous ones.
- Watch out for port conflicts: two applications cannot deploy inbound endpoints bound to the same ports.

Sharing Custom Code

Sharing Custom Code

Besides all the common code that exists in a company, there are Mule specific programmatic artifacts that are worth considering sharing.

Let's name a few:

*Custom transformers - performing operations that none of the Mule stock transformers can perform (see: http://www.mulesoft.org/documentation/display/MULE3USER/Creating+Custom+Transformers),
*Custom components - typically Mule-aware or non-business oriented components, as business components are usually simple POJOs coming from pre-existing projects (see: http://www.mulesoft.org/documentation/display/MULE3USER/Developing+Components),
*Custom expression evaluators - for the cases when a specific message extraction capacity is needed in several places of an application (see: http://www.mulesoft.org/documentation/display/MULE3USER/Creating+Expression+Evaluators).

The most convenient way to share custom code across team is to rely on Maven's dependency management mechanism. Here is an extract of a pom.xml referring to common code stored in a shared Maven library:

```
<dependency>
  <groupId>com.acme.mulings</groupId>
  <artifactId>common-mule-project</artifactId>
  <version>1.3</version>
</dependency>
```

The Mule build plug-in will automatically bundle these extra dependencies in the /lib directory of the deployable application. In this case, the common-mule-project-1.3.jar will be added to this directory at build time, ready to be deployed and made available to the application running on Mule.

Sharing Custom Configuration Fragments

Sharing Custom Configuration Fragments

Thanks to its element naming and referencing strategy, the Mule configuration configuration mechanism supports re-using fragments of configuration between teams. This is very convenient for sharing complex, repetitive or critical bits of configuration.

Concretely, here are some type of configuration elements that can be shared across projects:

*Connector configurations - a connector that need a complex configuration, say with specific transport level details, is a great candidate for re-use
*Endpoint definitions - defining global endpoints facilitates testing (as discussed in section 3) but also promotes re-use as they become sharable
*Pre-configured transformers - some transformers, like the XSL-T one, can require a fair bit of configuration, making them likely to be shared
*Sub-flows - a particular chain of message processors can form a bit of configuration that is worth re-using (we'll look at an example right after this)
*Flows - it can make sense for a service to exist in several Mule applications: in that case sharing a flow is the best way to go. If this flow is configured to rely on global endpoints, the application using this flow retains full control on what protocol will actually be used

Let's take a look at a configuration file containing a sub-flow that defines a standard chain of transformer we want to share with all our applications so they can use it too:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/core
http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd">

  <flow name="DefaultTransformers">
    <append-string-transformer message=">>" />
    <base64-encoder-transformer />
  </flow>
</mule>
```

As you can see, nothing makes this flow a sub-flow except that it has no inbound router. It's when if gets re-used that it truly becomes a sub-flow:

```
<flow name="FlowUsingSubflow">
  ...
  <log-component />
  <flow-ref name="DefaultTransformers" />
  ...
</flow>
```

Now, how does can an application refer to a common flow like this? In truth, what we're sharing are complete configuration files. We do so with the standard Maven dependency mechanism described before: the Maven project containing common configuration files (containing common configuration fragments) simply needs to export them as resources in a JAR.

The only (simple) trick is that the Mule application that wants to use a shared configuration file should import it by locating it on the classpath, as shown here:

```
<spring:import resource="classpath:mule-common-config.xml" />
```

Sharing Custom Configuration Patterns

Sharing Custom Configuration Patterns

Mule supports the notion of configuration patterns that are specialized flow tailored to serve a very specific purpose in the least amount of XML.

The list of configuration patterns that ship with Mule are listed here:
http://www.mulesoft.org/documentation/display/MULE3USER/Configuration+Patterns

Mule also provides the necessary tooling for creating a pattern catalog and patterns within it, allowing teams to encapsulate domain specific knowledge in custom configuration elements and share it across teams.

In essence, a pattern catalog is not different than a standard module: teams sharing pattern catalogs will only need to add a new dependency in their pom.xml. Of course, unlike what it's done with Mule standard modules, the dependency scope will have to be "compile" and not "provided."

Sharing Applications

Sharing Applications

Applications are self-contained archives and can therefore be shared via a simple file or web server.

You must document the name and content of the override properties file that each application is expecting.

Sustainable Software Development Practices with Mule

Sustainable Software Development Practices with Mule

The following describe sustainable software development practices with Mule.

Reproducible Builds

Reproducible Builds

You should be able to build a particular version of your Mule project at any point of time. This facilitates maintenance as the versions of your projects in production will certainly not be the latest ones.

The following will help you achieve this goal:

-
  - Source control all your Mule project, like any other project (remember not to check-in Eclipse specific files). Branching, merging and tagging are all applicable practices to Mule projects
  - Manage dependencies strictly: using Maven and an in-house repository manager (like Nexus) will get you a long way there

Continuous Integration

Continuous Integration

With all the previous emphasis on testing, setting up continuous integration for your project should look like a no-brainer. By using Maven as your build tool, you'll be able to set-up a build that gets triggered on every project change and run all its unit and functional tests automatically.\

There are plenty of continuous integration tools out there. To name a few: Hudson, TeamCity and Bamboo are popular choices.

If you've opted for using real transports in your functional test cases, you will have to pay attention to potential port conflicts that could occur in your continuous build server. Mule provides a convenient utility that can locate available ports and make them available as properties that your test specific transport configuration file can use. Read more about this tool here:
http://www.mulesoft.org/documentation/display/MULE3USER/Using+Dynamic+Ports+in+Mule+Test+Cases

If your target deployable is a web application and not a Mule application, consider using Jitr (http://jitr.org) for running your functional tests and avoiding port conflicts.

Repeatable Deploys

Repeatable Deploys

As mentioned in 2.2, it is highly desirable that your build produces the same deployable unit (ie. Mule or Web application depending on your deployment model) for all target environments. This is made possible by the externalization of all the configuration variables in properties files.

If the use of external overrides is not an option for you and you have to create different deployable units for your different environments, avoid manual operations at all cost. A good approach is to use a different Maven profile for each of your environments in order to control the build in a strict and reproducible manner.