Ajax IN ACTION

Dave Crane
Eric Pascarello
with Darren James

MANNING

Table of Contents

Preface

Part 1: The Groovy language

1 Your way to Groovy
2 Overture: The Groovy basics
3 The simple Groovy datatypes
4 The collective Groovy datatypes
5 Working with closures
6 Groovy control structures
7 Dynamic object orientation, Groovy style
8 Groovy Meta Programming

Part II: Around the Groovy library

9 Working with builders
10 Working with the GDK
11 Database programming with Groovy
12 Integrating Groovy
13 Working with XML
14 Unit testing with Groovy

Part III: Everyday Groovy

15 Tips and tricks
16 Groovy modules and frameworks
17 Concurrent programming with Groovy
18 Creating Domain Specific Languages
19 Groovy best practice patterns
Appendix A: Installation and documentation
Appendix B: Groovy language info
Appendix C: GDK API quick reference
Appendix D: Cheat sheets

2 Configuring Mule

In this chapter

- Communicating with the MuleClient
- Using context objects to interact with Mule
- AOP with component interceptors
- Leveraging the notification framework

In this chapter, you'll learn the fundamental principles of a Mule configuration file. Said differently, this chapter will give you the grammar of the configuration file, while the upcoming chapters will help you build your vocabulary. When you're done reading it, you'll be able to create new configuration files and set up the scene for your own services, which you'll learn to create in the coming chapters.

In essence, configuring Mule consists of defining the services you want to be active in a particular instance of the ESB. As seen in the previous chapter, these services are composed of and rely on many different moving parts, which also need to be configured. Some of these moving parts are intrinsically shared across several services, such as connectors. Others can be locally defined or globally configured and shared, such as endpoints. As you can guess, supporting this flexibility and richness in a configuration mechanism is pretty hairy.

To achieve this, Mule uses *configuration builders* that can translate a human-authored configuration file into the complex graph of objects that constitutes a running node of this ESB. The main builders are of two kinds: a Spring-driven builder, which works with XML files, and a script builder, which can accept scripting language files.

NOTE **The scripting configuration builder**

The scripting configuration builder uses a script in any language for which a JSR-223 compliant engine exists (such as Groovy, JRuby, or Rhino). By default, Mule supports Groovy configuration files, but other scripting languages can be added by installing the Mule Scripting Pack that can be downloaded from `http://mulesource.org/display/MULE/Download`. A scripted configuration is a low-level approach to configuring Mule. It's up to you to instantiate, configure, and wire all the necessary moving parts. This requires an expert knowledge of Mule internals. This explains why this approach is much less popular than XML configuration, as it's rare that anyone needs this level of control. This said, there can be circumstances where using Spring or XML isn't an option. In that situation, using a scripted configuration can save the day.

In this chapter we'll mainly focus on configuring the Spring XML builder, for several reasons:

1. *It's the most popular* you're more likely to find examples using this syntax.
2. *It's the most user friendly* Spring takes care of wiring together all the moving parts of the ESB, something you must do by hand with a scripted builder.
3. *It's the most expressive* dedicated XML schemas define the domain-specific language of Mule, allowing you to handle higher-level concepts than the scripting approach does.

We'll start by running a simple example, which will allow you to look at your first service configuration. We'll then review the overall structure, element families, and configurable items of the Spring XML file in general terms, just enough for you to get the gist of it and have the basic knowledge you'll need to grasp what you'll learn in the upcoming chapters. We'll also give some advice on how to organize your configuration files efficiently.

Are you ready for the ride? We believe the answer is yes, so let's start to look into our first Mule configuration file.

2.1 First ride

Though Mule comes complete with a 'hello' example, we chose to get started with the 'echo' example that's also bundled with the platform. We believe the echo example is the true 'hello world' example for Mule, as it doesn't require any transformer, custom component, or specific router. That's why we decided to nickname it 'Echo World .'

Let's now look into details of the Echo World example . As shown in figure 2.1, this application uses the `stdio` transport to receive messages from the console input (`stdin`) and send these messages unchanged directly to the console output (`stdout`) . A specific component, called a *bridge* , is used to pass the messages from the inbound router to the outbound one. A bridge is a neutral component: it doesn't perform any action or modify the messages that it processes. The outbound router is a pass-through one : it's the simplest router that exists. It dispatches the messages it receives to a unique endpoint.

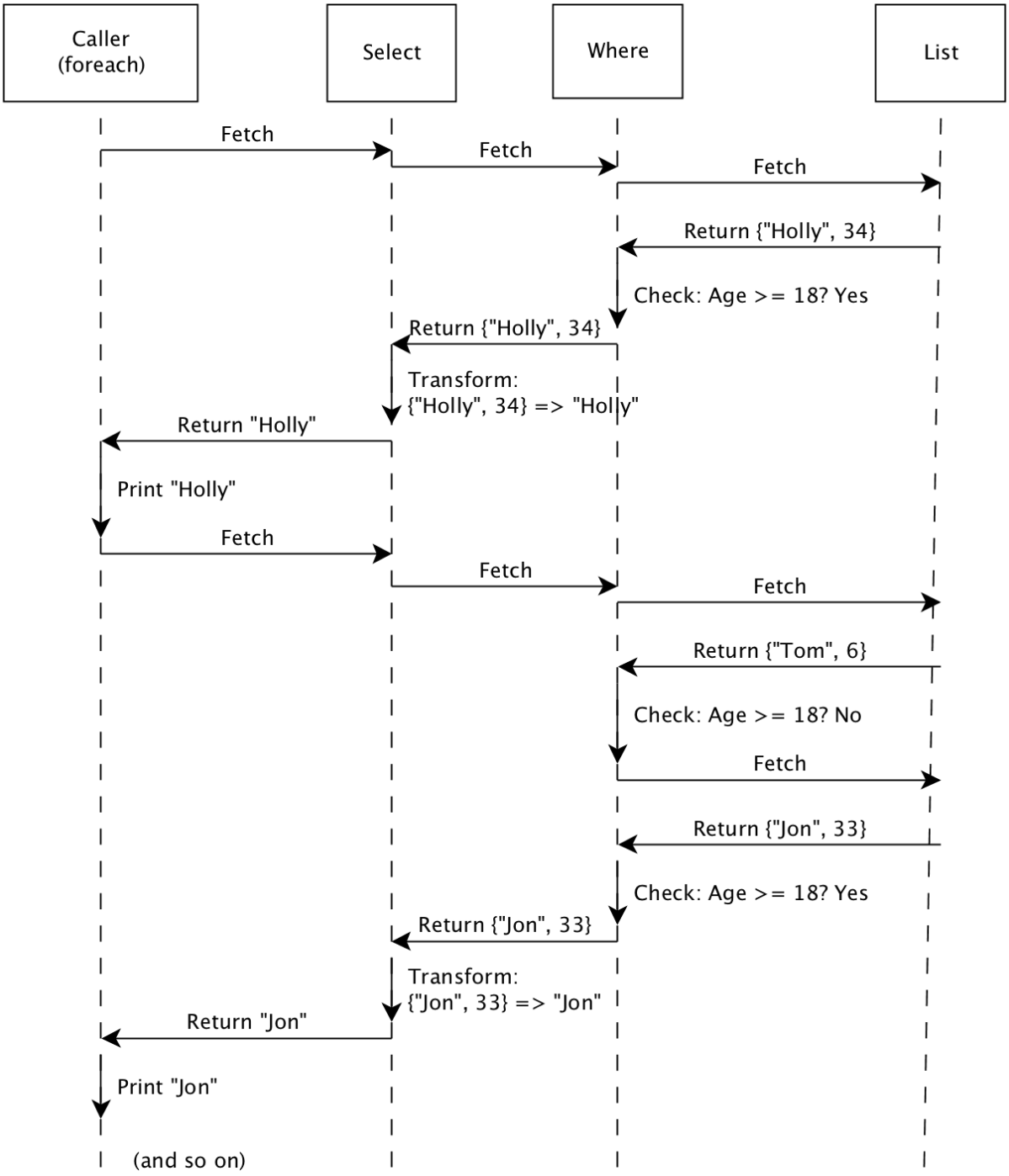


Figure 2.1 Moving parts and message flow of the echo example

Listing 2.1 provides the full configuration for this example. As an astute reader, you'll quickly notice that there's no bridge component. This is because a service uses a bridge component implicitly if none is configured. Except for this subtlety, the configuration file is pretty straightforward. Note how the specific attributes on the different elements help to make the configuration self-explanatory.

Listing 2.1 The echo example XML configuration

```
Source does NOT exist: /Users/antoine/Documents/CDI/cdi-book/AAMakePDF/codelistin
```

In example ❶, we installed the Mule standalone server and at ❷ downloaded the examples from this book's companion web site. So, at this point, you should be able to start this example. For this, go into the /chapter02/echo directory and run the echo-xml batch file that suits your OS. The output should be as shown here, with a version and a build number that match your Mule installation and with the host name of your machine :

```
Source does NOT exist: /Users/antoine/Documents/CDI/cdi-book/AAMakePDF/codelistin
```

As prompted, enter something. Note that the first time a message gets dispatched to the outbound endpoint, Mule instantiates and connects the necessary dispatcher:

```
Source does NOT exist: /Users/antoine/Documents/CDI/cdi-book/AAMakePDF/codelistin
```

When you're tired of playing with the echo, stop the execution of the application with `control+C`. All the log entries you'll see scrolling on your console will represent a normal and clean shutdown of the ESB . The very last lines shown in the console should be like this:

```
************************************************************************
* The server is shutting down due to normal shutdown request          *
* Server started: 28/02/09 15:52                                      *
* Server shutdown: 28/02/09 15:53                                     *
************************************************************************
<-- Wrapper Stopped
```

Note how `control+C` has been interpreted as a normal shutdown request. If the JVM were to exit abruptly, for example because of a hard crash, the wrapper script would restart the instance automatically.

Even if basic, this example has shown you what less than 20 lines of configuration can buy you in Mule. You also got the gist of the runtime environment of Mule, how it behaves at startup and shutdown times, and what it reports in the logs.

The Echo World example has given you some clues about the organization of the Spring XML configuration file.[1] Let's now broaden your view by exploring the structure of this configuration file.

Footnote 1 The same example is also provided as a scripted configuration in echo-world/conf/echo-config.groovy if you're curious to discover this configuration approach as well.

2.2 The Spring XML configuration

The Spring configuration builder relies on XML to enforce the correct syntax (well-formedness) and on XML Schema to define the grammatical rules (validity). These rules define the usable elements[2] and where they can be used. These elements represent the different moving parts of the ESB and their configurable parameters.

Footnote 2 *Element* as defined by the W3C DOM specification.

Mule doesn't rely on a single monolithic schema to define all the configurable elements, but instead on several of them . One of these schemas defines the core elements and the general structure of the configuration file. Other elements are defined by optional additional schemas, such as the transport-specific ones. If you look at the first lines of listing 2.1, you'll see that the core and the VM transport specific schemas are imported.

TIP	**Best practice** Always validate your XML configuration files before attempting to load them in Mule to simplify troubleshooting.

Figure 2.2 represents the elements defined by the core schema . Note how the services, which are the main actors of a configuration, are lost in the bottom-right corner of the mind map. This gives you an idea of all the supporting common features and global elements you can declare in a configuration file. Throughout the book, you'll progressively learn more about each of these elements.

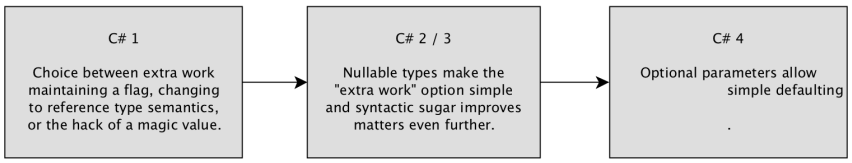


Figure 2.2 Configuration structure defined by Mule's XML core schema

The Echo World configuration in listing 2.1 only has a few schema references and one global element (a connector). You can easily imagine that real-world configuration files declare many of these shared configuration artifacts to support the configuration of their services. In the examples coming in the next chapters, you'll often see such shared elements, like global endpoints or transformers. Some chapters will even be dedicated to some of these common elements, such as transaction managers or JMX agents.

For now, we'll consider these elements under several of their main characteristics. First we'll look at their main families, then how we can define configured values, and finally where to find the schemas that define them.

2.2.1 XML element families

Independent of their functions, we can conceptually distinguish three main families of elements in a Mule configuration:

- The specific elements
- The custom elements
- The Spring elements

It's important to understand what you can expect from each of these to build a configuration file in the most efficient manner.

Specific elements

The specific elements are the most expressive configuration elements available, insofar as they constitute the domain-specific language of Mule . As such, these elements and their attributes are characterized by highly specific names. Another characteristic is that they intrinsically refer to predefined concrete implementations: they're used to express all the configuration elements that aren't custom-made. This is why no class name comes into play when using this kind of configuration element. A benefit of this is the isolation your configuration gets from the internals of Mule's implementation: if a class is renamed, your configuration won't be affected.

The Mule core and each transport or module you include in your configuration will contribute such specific elements. Table 2.1 is a nonexhaustive list[3] of these elements as defined by a few Mule XML schemas.

Footnote 3 There are more XML elements defined in these schemas; only the most notable ones are listed here.

Table 2.1 Specific elements defined by some Mule schemas

Schema	Elements defined (nonexhaustive list)
Mule Core	• *Models* • *Transformers* • *Routers* • *Filters* • *Components* • *Security manager* • *Exception strategies* • *Transaction manager*
HTTP Transport	HTTP connector HTTP endpoint REST service component HTTP request and response transformers

XML Module	XML transformers XML routers XML filters

Custom elements

Custom elements constitute the Mule-oriented extension points of a configuration. They mainly allow you to use custom implementations or Mule classes for which no specific element has been created. These extension points are available for core Mule artifacts and also for some transports and modules. Custom elements usually rely on a fully qualified class name parameter to locate your custom code. They also provide a means to pass the property values your class needs to be properly configured. Table 2.2 shows a few examples of these custom elements.

Table 2.2 Custom elements defined by some Mule schemas

Schema	Elements defined (nonexhaustive list)	Example
Mule Core	Custom transformer Custom router Custom filter Custom entry point resolver Custom security and encryption providers Custom exception strategy Custom transaction manager	`<custom-transformer name="NameStringToChatString" class= "org.mule.example.hello.NameStringToChatString"/>` Defines a custom transformer named `NameStringToChatString` as an instance of the specified class.
TCP Transport	Custom protocol	`<tcp:custom-protocol class="org.mule.CustomSerializationProtocol"/>` Defines a custom TCP protocol as an instance of the specified class.

Though it's possible to do what a specific XML element can do using a custom XML element, this should be avoided for two main reasons. The first is because you'll become coupled to a particular implementation in Mule itself: if a new class is created and used by the specific element, your custom one will keep referencing the old one. The second is because you'll lose the benefit of the strongly typed (schema validated) and highly expressive attributes defined on the specific elements.

TIP **Best practice**
If possible, try to use a specific configuration element instead of custom one.

SPRING ELEMENTS

Thanks to the support of a few core Spring schemas, Mule can accept Spring elements in its configuration. Spring elements are used to instantiate and configure any object that'll be used elsewhere in the configuration, where they'll usually be injected in standard Mule elements. They're also used to easily construct configuration parameters such as lists or maps. Finally, they allow configuration modularity through the support of the import element (see section 2.3). Table 2.3 shows a few examples of these Spring elements. Note how the `spring:beans` element opens the door to the usage of any Spring schema (in this example we use the `util` one).

Table 2.3 Elements defined by the supported Spring schemas

Schema	Elements defined (nonexhaustive list)	Example

Spring Beans	Beans
	
	Bean
	
	Property
		`<spring:bean name="cfAMQ"`
		`  class=`
		`  "org.apache.spring.ActiveMQConnectionFactory">`
		`  <spring:property name="brokerURL"`
		`  value="tcp://localhost:61616"/>`
		`</spring:bean>`
		
		Instantiates an ActiveMQ connection factory, configured with the specified broker URL parameter and named `cfAMQ`.
		
		`<spring:beans>`
		`  <spring:import`
		`  resource="classpath:applicationContext.xml" />`
		`</spring:beans>`
		
		Imports a Spring application context from the classpath.
		
		`<inbound-endpoint ref="globalEndpoint">`
		`  <properties>`
		`  <spring:entry key="valueList">`
		`  <util:list>`
		`  <spring:value>value1 and others</spring:value>`
		`  <spring:value>value2</spring:value>`
		`  </util:list>`
		`  </spring:entry>`
		`  </properties>`
		`</inbound-endpoint>`
		
		Sets a list property on an endpoint.

Spring Context	Property placeholder resolver	`<context:property-placeholder`
		`  location="node.properties"/>`
		
		Activates the replacement of placeholders, resolved against the specified properties file.

When you start writing your own configuration files, you'll quickly realize how

valuable this is. Whenever you configure a Mule object, the question of its default configuration values will arise. What would usually take a visit to the JavaDoc page of the object will now require only a glance at the default values for the different attributes supported by the XML element that represent the object (provided you use a decent tool).

A SECT3 SHOULD COME LAST

Besides configuration extension, Spring elements also grant access to the framework itself, making possible the usage of advanced features such as AOP. Though you can do a lot in Mule without deferring to Spring, getting acquainted with this framework will allow you to better grasp the core on which Mule is built. If you're unfamiliar with Spring or would like to learn more about it, we strongly recommend reading *Spring in Action* from Manning Publications Co. (Walls and Breidenbach).

You've now learned to recognize the different families of XML elements you'll have to deal with and what you can expect from them. XML elements without values specific to your configuration are useless. Let's now see how you'll set these different values in your own configuration files.

2.2.2 Configured values

In a Mule configuration, values are seldom stored as text nodes; they're mainly stored as attributes of XML elements. The main exceptions are text-rich properties such as documentation or script elements. This makes the configuration easier to read, format, and parse, as attributes are less prone to being disrupted by unwanted whitespaces. These attributes are strongly typed, with data types defined by the XML Schema standard.

We'll consider these configured values under their most notable traits:

- Default values
- Enumerated values
- Expressions
- Property placeholders
- Names and references

DEFAULT VALUES

Using attributes to hold values enables the definition of default values. Mule schemas take great care to define default values wherever possible. When you start writing your own configuration files, you'll quickly realize how valuable this is. Whenever you configure a Mule object, the question of its default configuration values will arise. What would usually take a visit to the JavaDoc page of the object will now require only a glance at the default values for the different attributes supported by the XML element that represent the object (provided you use a decent tool; see chapter XREF dev.ch).

ENUMERATED VALUES

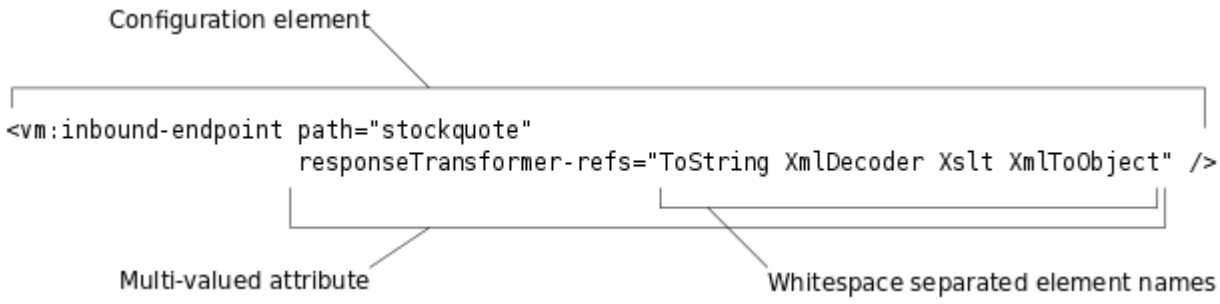
On top of being strongly typed, some attributes define an enumeration of possible configuration values . This greatly reduces the risk of introducing a bogus value that would only be caught later on at runtime. This also provides valuable information about the intent of the parameter and the abilities of the configured object.

EXPRESSIONS

Mule's expression evaluation framework offers the capacity to define configuration values that are evaluated dynamically at runtime. With simple attribute values of the form `#[evaluator:expression]`, it is possible to access almost any data from the current message being processed or the Mule instance itself. Please refer to appendix XREF expreval.anx for more information about this powerful framework.

PROPERTY PLACEHOLDERS

An important aspect of any configuration file is the capacity to externalize certain values that can change at runtime. These values are called *properties* . They're generally used to define environment-specific parameters such as credentials, port numbers, or paths. It's possible to use such properties in lieu of fixed values in any attribute of a Mule configuration. The property is then referred to by its name using a special placeholder syntax. This is done using the classic Ant notation , as shown:



These properties are defined either as global ones in the Mule configuration , in a standard Java properties file or in JVM system properties . For the last two options, Spring's property placeholder resolver (shown in table 2.3) takes care of injecting the properties value in your configuration.

TIP **Environment properties**

As with any other application, you'll have to externalize environment-specific values in properties files. These are usually passwords, remote service URLs, port numbers, time-outs, or cron expressions. With Spring XML configurations, a good practice is leveraging Spring's `PropertyPlaceholderConfigurer` to load properties from several files on the classpath. The idea is to define reasonable defaults for the development environment in a properties file embedded in the deployable itself, and to override these values with others defined in an environment-specific property file placed also on the classpath, but outside of the deployable. This is demonstrated here:

```
<context:property-placeholder
              location="classpath:META-INF/default.props, [CA] \
              classpath:override.props" />
```

This configuration element imports values from classpath files META-INF/default.props and from override.props. The latter can be left empty in development and tuned to use correct values in test or production environments. This construct also supports overriding with Java system properties: if a system property is defined, it'll take precedence over a property of the same name defined in one of these files.

NAMES AND REFERENCES

Being able to have elements that reference other elements is an essential aspect of a Mule configuration file. This is achieved by using name and reference attributes . Most of the elements can receive a name through this mechanism. References can sometimes be multivalued. In that case, whitespace is used to separate the different names that are referred to. The following demonstrates an inbound endpoint that refers to a chain of four transformers simply by listing their names:

You now know the secrets to setting values in your XML configurations. We'll now look at one thing that may still puzzle you: the location of all these different schemas.

2.2.3 Schema locations

You might be wondering where to get an up-to-date list of all the available schemas you can use in your configurations. This section will do better than give you this list: it'll allow you to figure out by yourself the right schema reference to use for any Mule library you decide to use.

When you look at the declaration in listing 2.1, you might be wondering if Mule will connect to the Internet to download the different schemas from the specified locations. Of course it doesn't. Each library (transport or module) embeds the schemas it needs . Mule leverages the resource resolver mechanism of Spring to 'redirect' the public HTTP URIs into ones that are internal to the JAR file . How is this mechanism configured? To discover it, fire up your favorite archiving utility or IDE and open the library you want to use. There should be a directory named META-INF. After opening it, you should see something similar to the screen shot shown in figure 2.3.

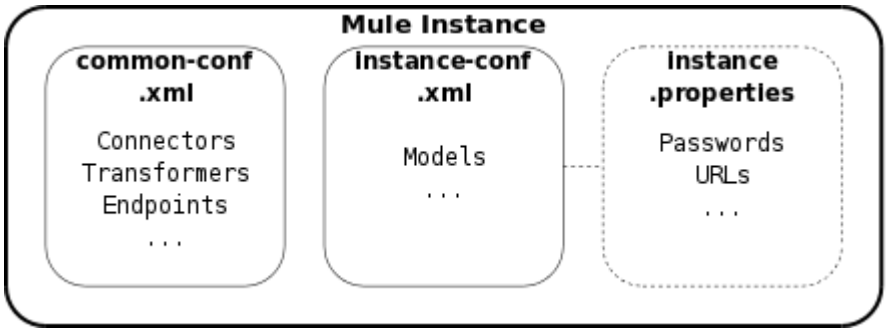


Figure 2.3 Locating the XML schema of the VM transport

The schema you're looking for is in the META-INF directory and is named after the transport or module name, prefixed with `mule-`. Figure 2.3 shows that in the VM transport, the schema is named mule-vm.xsd. Some transports have several schemas, one per variation of the main transport (such as HTTP and HTTPS). The target namespace of the schema you're looking for is declared on the root element. Listing 2.3 shows the root element of the VM transport schema.

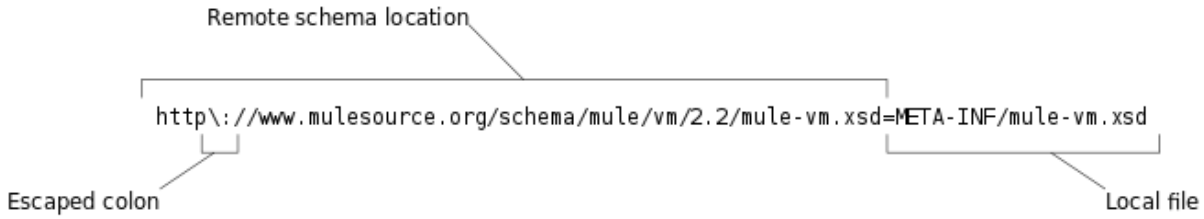
Listing 2.2 The target namespace to use in configuration files is defined in the schema of each Mule library.

```
Source does NOT exist: /Users/antoine/Documents/CDI/cdi-book/AAMakePDF/codelistin
```

Listing 2.3 This example tests the EL embedded

```
Source does NOT exist: /Users/antoine/Documents/CDI/cdi-book/AAMakePDF/codelistin
```

By now you should be anxious to know where the schema location is defined. Using the right location is important in order for the 'redirection' mechanism to kick in and allow Mule to validate and load your configuration file. This mechanism is configured by the file named spring.schemas, which is located alongside the library schema. You can see this file in figure 2.3. The content of this file is a simple mapping between the remote schema location and the archive file to use for it, as shown:



The schema location you want to use is on the left of the equal sign. Bear in mind that because properties files require you to escape colons, the backslash in front of the equal sign should be omitted if you copy and paste the location directly into your configuration file. With the target namespace and the schema location in hand, you should now be able to add the VM transport schema to your configuration file. Listing 2.4 shows you what you should have come up with.

Listing 2.4 The root element of a Mule configuration that uses the VM transport

```
Source does NOT exist: /Users/antoine/Documents/CDI/cdi-book/AAMakePDF/codelistin
```

Throughout this chapter, we've referred to the Mule core schema. If you look for it in mule-core.jar , you'll be disappointed. It's in fact located in mule-module-spring-config.jar . Why there and not in the core JAR? Because using Spring XML is one possibility for building a Mule configuration. As we said in the introduction, there are other ways, such as scripting.

TIP **My valid configuration doesn't load!**

It's possible to create a configuration that refers to all the right schemas and is well formed and valid, but still doesn't load. If this happens to you and you get a cryptic message like the following :

```
Configuration problem: Unable to locate
NamespaceHandler for namespace
[http://www.mulesource.org/schema/mule/xyz/2.2]
```

Then get ready for a 'duh' moment. This means you've forgotten to add to your classpath the Mule library that defines the namespace handler required to load configuration elements in said namespace. Watch out for missing transport or module JARs.

By now, you're certainly eager to rush to your keyboard and start building your own configurations. Before that, we'd like to introduce the notion of modular configurations. This will help you tame the complexity that may arise in your configurations when your projects start expanding.

SIDEBAR **Another hint**

Consider also reading chapter, where we discuss the development tools you'll need, before starting a Mule project.

2.3 Configuration modularity

Have you ever had to wade through pages and pages of configuration, trying to sort things out and find your way around? As your integration projects grow and multiply, there's a risk that your configuration files may become bloated or redundant, hence hard to read, test, and maintain. Fortunately, the configuration mechanism of Mule allows you to relieve the ails of a monolithic configuration by modularizing it .

IMPORTANT **Best practice**

Modularize your configuration files to make them easier to read and simplify their maintenance.

Cutting a configuration into several parts encourages the following :

- Reuse of common artifacts across several configurations
- Extraction of environment-dependent configuration artifacts
- Isolation of functional aspects that become testable in isolation

There are different approaches that can be used when modularizing a configuration. In the following sections, we'll detail these strategies:

- Independent configurations
- Inherited configurations
- Imported configurations
- Heterogeneous configurations

Of course, you can combine them together. At the end of the day, the objective of this discussion is to give you a hint of the possibilities and let you establish the configuration organization that best fits your project size and needs.

WARNING	**Using an existing context** In section 7.1, we'll discuss another handy option: using an existing Spring context as the parent context of your Mule configuration. Because this isn't done at configuration level, we won't detail this approach here.

2.3.1 Independent configurations

As shown in figure 2.4, a Mule instance can load several independent configuration files side by side . In this example, Mule will use two Spring XML configuration builders to instantiate, configure, and wire together the elements defined in both XML configuration files. As you can see, the environment-dependent properties have been exported in an external file, loaded from the instance-specific configuration file.

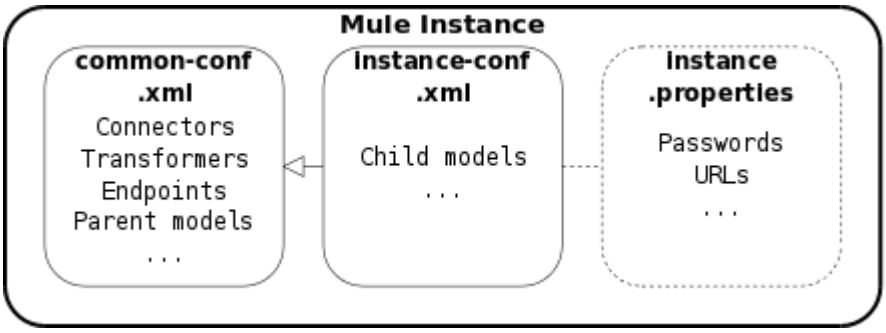


Figure 2.4 A Mule instance can load independent XML configuration files and properties files. We also want to test out a multi-line figure caption.

This approach is well suited for simple scenarios, where there's a loose coupling between the common and instance-specific elements. It allows a fair level of reuse, as global elements such as connectors, transformers, or endpoints can be shared

with several instance-specific configurations.

The main drawback of using independent configurations is that the instance-specific configuration doesn't strongly express its need to run alongside the common configuration. If the shared elements aren't used immediately, like transformers, you can start the Mule instance without the required common configuration and start having issues only later on at runtime.

Using inherited configurations alleviates this problem.

2.3.2 Inherited configurations

The concept of inherited configurations is illustrated in figure 2.5. The main idea is to express a formal parent-child dependency between two configurations . By strongly expressing this dependency, you'll have the guarantee at boot time that no configuration file has been omitted (unlike the behavior you get with side-by-side independent configurations as described in the previous section).

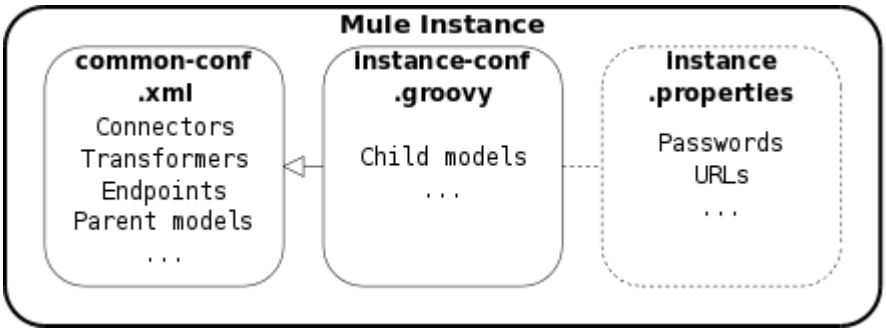


Figure 2.5 Models can be inherited, leading to enforced hierarchies of configuration files.

Inheritance can only be defined between models: this is why the parent and child models are represented. Because a model can be empty, this inheritance approach is suitable even if the common configuration doesn't have any model elements to share. It's also good to know that several different models can inherit from the same parent. The modularization can therefore span several configuration files.

How is this inheritance expressed? Simply by using the same name for the parent and child models and by flagging the child as being an heir , as shown here:

```
<model name="myConfig">
<model name="myConfig" inherit="true">
```

In this configuration sample, the parent model doesn't have an `inherit` attribute, which defaults to false when absent, and the child model has the attribute set to `true`.

2.3.3 Imported configurations

As briefly discussed in section 2.2.1, Spring can be leveraged to instantiate and configure any infrastructure or custom bean you need for your Mule instance. For example, Spring can perform JNDI lookups or wire your business logic beans together. After time, your Spring configuration may grow to the point that it starts to clutter your Mule configuration. Or you might decide that part of your Spring configuration is reusable. At this point, extracting your Spring beans to a dedicated application context file would be the right thing to do. Figure 2.6 shows a situation where such shared application contexts have been created and are used by a hierarchy of Mule configurations.

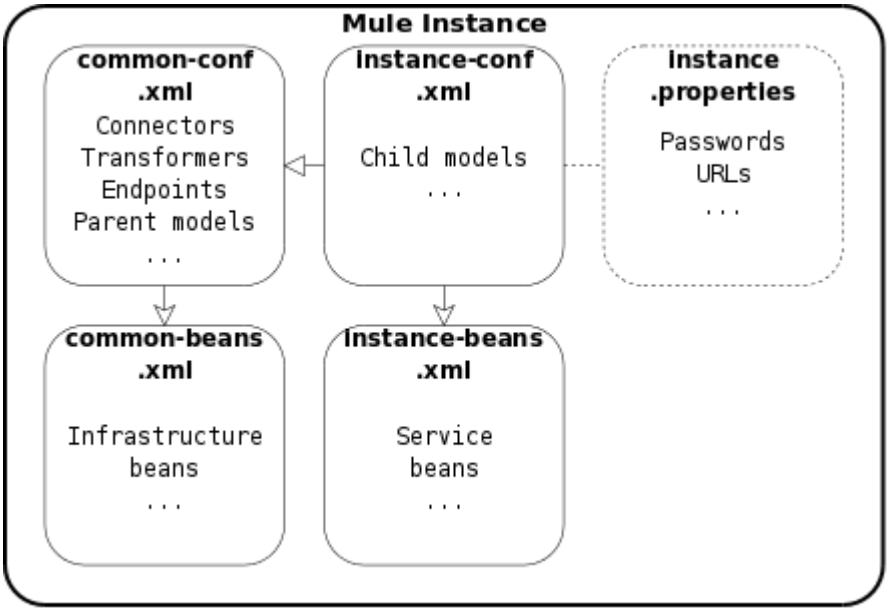


Figure 2.6 Imported Spring application context files can complement a hierarchy of Mule configurations.

You can easily import external Spring application context files from your Mule configuration files. [4] The following illustrates how `instance-conf.xml`, shown in figure 2.6, would import its Spring context file:

Footnote 4 Interactions between Mule and Spring configurations are further discussed in XREF ch07.embedded.sec.

```
<spring:beans>
    <spring:import resource="instance-beans.xml" />
</spring:beans>
```

Refer to Spring's documentation for more information on the usage of the import element.

2.3.4 Heterogeneous configurations

In the introduction to this chapter, we evoked the possibility of configuring Mule with a scripting language. The tedious work of setting up the core runtime environment and all the boilerplate code this requires is usually enough to put off the bravest script aficionado. The good news is that there's a solution. It's possible to mix several styles of Mule configuration in an instance. The example in figure 2.7 shows an instance that has been configured with a Groovy script and Spring XML configuration builders. By following this approach, you can declare all the global elements using the XML syntax and focus your scripted configuration on the services you need for your integration project.

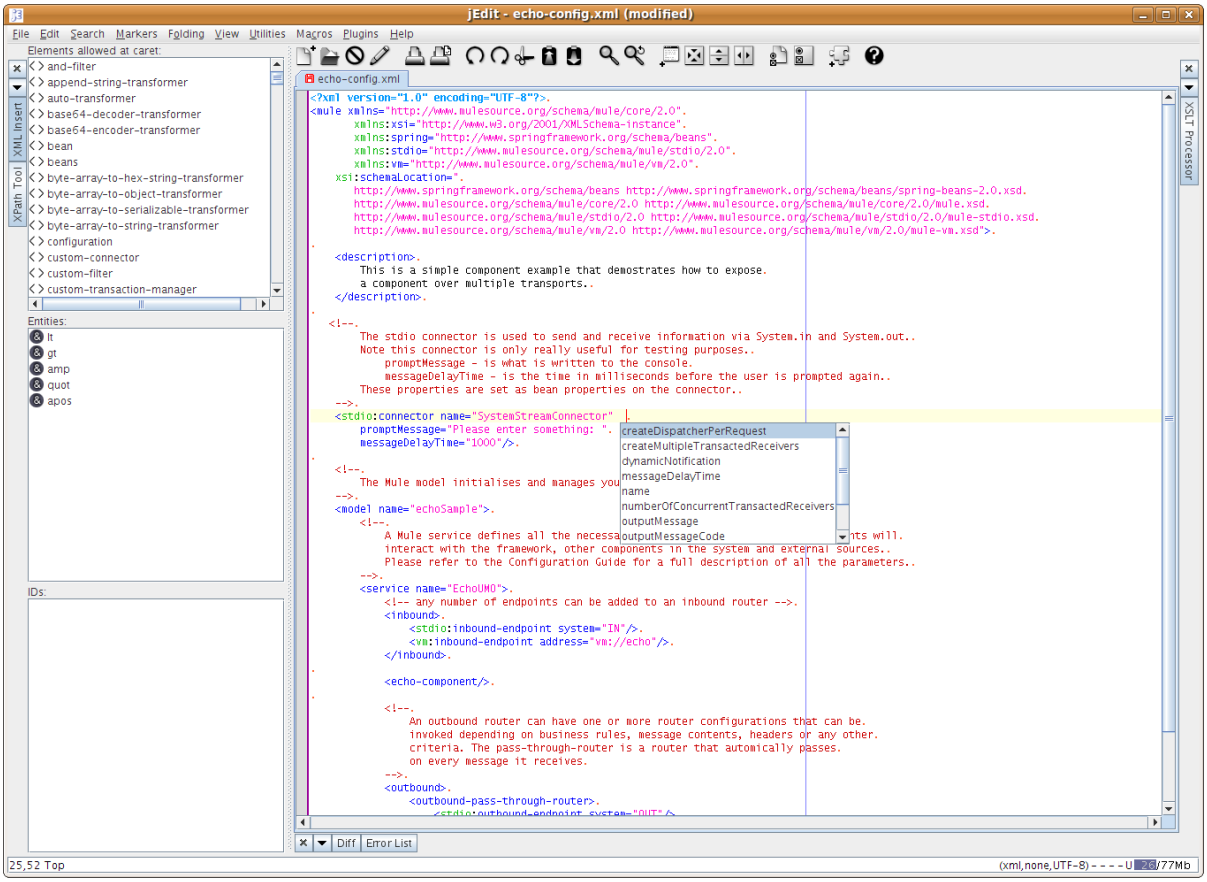


Figure 2.7 Mule can use heterogeneous builders to load a hierarchy of configuration files written in different syntax.

Since scripted configuration works at the bare metal level, you express the dependency to a parent by looking up the model you want to depend on. If in figure 2.7, the common configuration defined a model named `commonModel`, the way the Groovy configuration would look it up and use it would be the following:

```
model = muleContext.registry.lookupModel("commonModel")
...
childService.model = model
```

As you can see, there's no child model *per se*. It's up to the service in the child model to actively register itself in the model defined in the common configuration.

We're sorry to report that after reading this section, you have no more excuses for building monolithic and kilometric configuration files. And if you've inherited such configurations, you should now have some pretty good ideas on how you could refactor them.

2.4 Summary

In this chapter, you've learned the general principles involved in configuring Mule. You've discovered the overall structure of the Spring XML configuration, its syntax, and the ways to define properties values. Several strategies for organizing your configuration files have been detailed: they'll allow you to grow your integration projects without getting lost in lengthy and monolithic XML files.

The scripted configuration mechanisms have been rapidly covered as well. It's up to you to decide on your syntax of choice for configuring Mule. By looking at the examples at the end of this chapter, you should have a fair idea of the efforts involved in the XML and scripted configuration options. In the rest of the book, all the examples will use the Spring XML syntax.

In many places, this chapter has referred you to other chapters. Without this crafty stratagem, this chapter would've had the size of the whole book. The coming chapters will look further into the details of the main Mule moving parts: endpoints, routers, transformers, and components. You'll learn about the main types of each and all the good they can do for you.

IMPORTANT **Before we close...**

We'd like to introduce you to Clood, Inc. , our fictional but promising startup specializing in the domain of value-added services for businesses hosting applications in the cloud. Blessed with a double O, our startup is bound for a great future and a significant IPO. On the technical side of things, Clood, Inc., decided to leverage Mule as our integration platform to support all the services (monitoring, deployment controls, identity management, DNS, and so on) we intend to offer to our clients.

We're happy to inform you that, always looking for a challenge and some stock options, you've decided to join us on this cloudy adventure. Throughout the rest of the book, we'll review some of our activities at Clood, Inc., and how we're using Mule to accomplish them.

Index Terms

Ant notation
bridge component (implicit)
Clood, Inc. (presentation)
component (bridge)
configuration (builder)
configuration (scripted)
configuration (Spring XML) [advantages]
configuration (Spring XML)
configuration (XML schema)
configuration (XML schema) [core]
configuration (families of elements)
configuration (specific elements)
configuration (custom elements)
configuration (Spring elements)
configuration (data types)
configuration (default values)
configuration (enumerated values)
configuration (properties)
configuration (names and references)
configuration (XML schema) [transport]
configuration (XML schema) [in JAR files]
configuration (XML schema) [location]
configuration (XML schema) [namespace handler]
configuration (modularity)
configuration (modularity) [advantages]
configuration (reuse)
configuration (environment dependent)
configuration (testability)
configuration (modularity) [independent files]
configuration (modularity) [inheritance]
configuration (modularity) [importing files]
configuration (modularity) [mixed format]
configuration (Groovy)
console (reading from) [stdin]
console (writing to) [stdout]
custom-protocol
custom-transformer
domain specific language
echo world
global-property
Groovy
JSR-223 (See scripting)
model (inheritance)
Mule (basic example)
Mule (domain specific language)
mule-core.jar file
mule-module-spring-config.jar file
NamespaceHandler
object-to-string-transformer
pass-through-router
polling-connector
properties (global) [See also global-property]
properties (file)

properties (system)
properties (override)
PropertyPlaceholderConfigurer
property placeholders
scripting
Spring (XML configuration builder)
Spring (elements in configuration)
Spring (beans)
Spring (property)
Spring (import)
Spring (util schema)
Spring (property placeholder resolver)
Spring (resource resolver)
Spring (import)
Spring in Action
standalone server (startup sequence)
standalone server (stopping)
standalone server (shutdown sequence)
standalone server (wrapper script)
stdio (transport)
XML configuration (See configuration)
XML Schema (See configuration)
xslt-transformer