Progress FUSE

FUSE™ Mediation Router

Getting Started

Version 1.6
April 2009

PROGRESS SOFTWARE

Getting Started

Version 1.6

Publication date 17 Jul 2009

Legal Notices

Table of Contents

List of Figures

List of Examples

Chapter 1. Introducing FUSE Mediation Router

This chapter introduces the FUSE Mediation Router architecture and other basic concepts.

What is FUSE Mediation Router?

Overview

FUSE Mediation Router is an open-source integration framework. It enables you to define and implement *routes*—declarative solutions for specific integration problems. A route defines an integration path between two or more endpoints, a path from an input source to one or more output destinations. Each endpoint in a route, whether it is an input source or an output destination, is identified by a URL. FUSE Mediation Router supports a wide variety of endpoint types (and URLs).

You can deploy FUSE Mediation Router in a variety of environments, such as Tomcat servlet containers, OSGi containers, and J2EE application servers. FUSE Mediation Router works with the following FUSE products:

- FUSE ESB (Apache ServiceMix)
- FUSE Message Broker (Apache ActiveMQ)
- FUSE Services Framework (Apache CXF)

You also can use FUSE Mediation Router either on a standalone basis or in any Spring Framework-hosted application.

Enterprise integration patterns

Routes enable you to easily implement enterprise integration patterns (EIPs) using plain old Java objects (POJOs). To learn more, read the authoritative book, *Enterprise Integration Patterns*, by Gregor Hohpe and Bobby Woolf. The book describes a number of design patterns for the use of enterprise application integration and message-oriented middleware. Also see Enterprise Integration Patterns[1] and Apache Camel Enterprise Integration Patterns Guide[2].

Route building languages

FUSE Mediation Router provides two functionally equivalent ways to define routes:

- a Java domain specific language (DSL) — the DSL, also referred to as a *fluent API*, is a programming language designed for implementing enterprise integration patterns.

  One advantage of using a DSL is that the FUSE Mediation Router can support type-safe smart completion of routing rules in your IDE using regular

[1] http://www.enterpriseintegrationpatterns.com
[2] http://activemq.apache.org/camel/enterprise-integration-patterns.html

Java code without using extensive XML configuration. If you change your DSL rules, you just recompile your Java sources.

Example 1.1 on page 11 shows a DSL route that implements a Wire Tap pattern.

Example 1.1. DSL Wire Tap Pattern

```
from("direct:start")
   .to("log:foo")
   .wireTap("direct:tap")
   .to("mock:result");
```

- a Spring XML file — FUSE Mediation Router provides custom Spring elements that mimic the routing DSL. The advantage of using an XML file is that it changing a route requires no recompilation.

  Example 1.2 on page 11 shows an XML route that implements the same Wire Tap pattern shown in Example 1.1 on page 11.

  Example 1.2. Spring XML Wire Tap Pattern

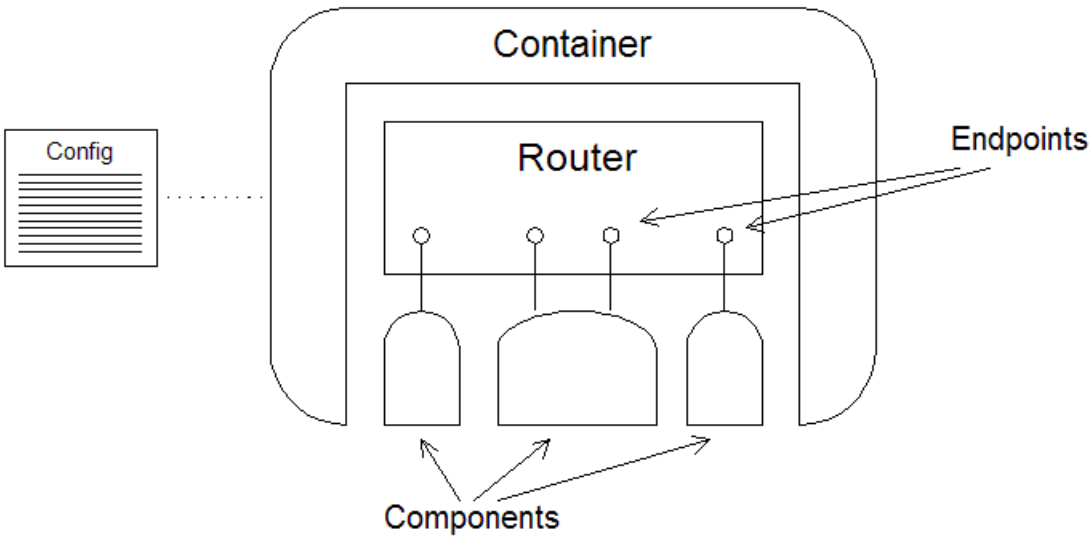
```
<route>
   <from uri="direct:start"/>
   <to uri="log:foo"/>
   <wireTap uri="direct:tap"/>
   <to uri="mock:result"/>
</route>
```

Architecture

Overview

Figure 1.1 on page 12 shows a general overview of the FUSE Mediation Router architecture. This architecture enables you to deploy across a wide variety of container types.

Figure 1.1. Architecture of the FUSE Mediation Router



Router

A router application is constructed using either Java DSL, XML configuration, or a combination of both. At runtime the router is represented by a `CamelContext` object. The `CamelContext` object encapsulates routing rules contained in `RouteBuilder` objects, and *components* that enable the router to bind to various network protocols and other resources.

Endpoints

An endpoint is a source or a sink of messages, identified by a URI. This means that an endpoint maps either to a network location, or to some other resource that can produce or consume a stream of messages. Within a routing rule, endpoints are used in two distinct ways:

- the *source endpoint* appears at the start of a rule in a `from()` statement.

  A source endpoint acts as a source of the messages processed by a route. If the route creates reply messages, the source endpoint acts as a sink for the replies.

- the *target endpoint* appears at the end of a rule in a `to()` statement. A target endpoint acts as a sink for messages. Target endpoints can also act as a source of reply messages.

Components

A component is a plug-in that integrates the router core with a particular network protocol or external resource. From the perspective of a route developer, a component appears to be a factory for creating a specific type of endpoint. For example, there is a file component that can be used to create endpoints that read/write messages to and from particular directories or files. There is an Apache CXF component that enables you to create endpoints that communicate with Web services.

Before you can use a particular component, you must configure it and add it to the route's configuration. You do not have to do this for the following components, which are embedded in the router core:

- Bean — Binds Java beans to message exchanges. This component is also useful for exposing and invoking POJO (Plain Old Java Objects).
- Direct — Provides direct, synchronous invocation of any consumers when a producer sends a message exchange. A direct endpoint can be used to connect existing routes. Clients running in the same JVM as the router can also access direct endpoints.
- File — Provides access to the file system, enabling you to read messages from files and write messages to files.
- Log — Logs the message exchange to some underlying logging system like log4j.
- Mock — Provides a powerful declarative testing mechanism similar to jMock. This component allows declarative expectations (assertions) to be created on any Mock endpoint before a test begins. When running tests, messages are fired to one or more endpoints and the assertions are checked.
- SEDA — Provides asynchronous SEDA behaviour, so that messages are exchanged on a BlockingQueue, and consumers are invoked in a separate thread from the producer.

- Timer — Generates message exchanges when a timer fires. This component can only be used to define consumer endpoints (appearing at the start of a route).
- VM — Enables asynchronous calls to another endpoint in the same JVM.

For more details about the available components see the *Deployment Guide* and the list of Camel components[3].

RouteBuilders

The `RouteBuilder` classes encapsulate the routing rules. A router developer defines custom classes that inherit from the base `RouteBuilder` class, and adds instances of these classes to the router's `CamelContext` object.

Deployment Options

The router architecture supports these deployment options:

- *Spring container deployment* — You can deploy the router application into a Spring container, using the Spring configuration file to configure components and define routes.
- *Standalone deployment* — You can write a `main()` method in the application code, which is responsible for creating and registering `RouteBuilder` objects, as well as configuring and registering components.
- *OSGI* — You can deploy the router application into an OSGI container.

For more details about the deployment options, see the *Deployment Guide*.

CamelContext

A `CamelContext` object represents a single FUSE Mediation Router rulebase. It defines the context used to configure routes and the details which policies should be used during message exchanges between endpoints.

[3] http://activemq.apache.org/camel/components.html

How to Develop a Router Application

Overview

Regardless of the complexity of the application being written there are two things a developer will always need to consider:

- the routing language to use
- the application's deployment environment

The basic steps for developing a router application are also the same no matter the complexity of the problem.

Choosing a routing language

One of the biggest choices faced by a FUSE Mediation Router developer is whether to use the Java DSL or the Spring XML to define the routes. In most cases the Java DSL is a better choice than the Spring XML for the following reasons:

- It is possible to configure an IDE's content completion feature to work with the Java DSL.
- It allows for the mixing of normal Java code with the Java DSL and therefor provides a richer language.
- It keeps all of an applications logic in a single format and not split between code and configuration.
- It is easier to manage than a large amount of XML.

Using the Spring XML to define routes does offer some flexibility because the router application does not need to be recompiled everytime a route is changed. It is also a nice environment for developers who are not comfortable with Java code.

Choosing the deployment environment

The choice of deployment environment is typically out of the hands of an individual developer. The environment an application is ultimately deployed in determined by factors such as corporate standards and large grained project requirements.

Fortunately, FUSE Mediation Router has a flexible deployment model and the core part of an application, the routing rules, are not effected by how the application will be deployed. A developer could write and do a large part of the testing using a standalone deployment. When the time comes to deploy

the router into it ultimate deployment environment it is a simple matter of removing the `main()` method used to instantiate the standalone router application.

Development steps

To develop a router application the following high-level steps are involved:

1. Define routing rules using either Java DSL, XML, or both.
2. Implement additional business logic using POJOs.
3. Configure non-core components.
4. Deploy the router.

   See *Deployment Guide*.

Chapter 2. Creating a Simple Content-Based Router

It is easy to create a simple content-based router using the Apache Maven tooling and the Java DSL.

Prerequisites

Overview

The following are required to complete this example:

- Internet connection (required by Maven)
- "Java Runtime"
- "Apache Maven 2"

Java Runtime

FUSE Mediation Router requires a Java 5 development kit (JDK 1.5.x). After installing the JDK, set your `JAVA_HOME` environment variable to point to the root directory of your JDK, and set your `PATH` environment variable to include the Java `bin` directory.

For more information, see *Installation Guide*.

Apache Maven 2

The FUSE Mediation Router Maven tooling requires Apache Maven version 2.0.6 or later. To download Apache Maven, go to http://maven.apache.org/download.html.

After installing Apache Maven do the following:

1. Set your `M2_HOME` environment variable to point to the Maven root directory.
2. Set your `MAVEN_OPTS` environment variable to `-Xmx512M` to increase the memory available for Maven builds.
3. Set your `PATH` environment variable to include the Maven `bin` directory:

Platform	Path
Windows	`%M2_HOME%\bin`
UNIX	`$M2_HOME/bin`

Tip

You can use the maven command **mvn -version** to verify that Java and maven are installed correctly. The example output is:

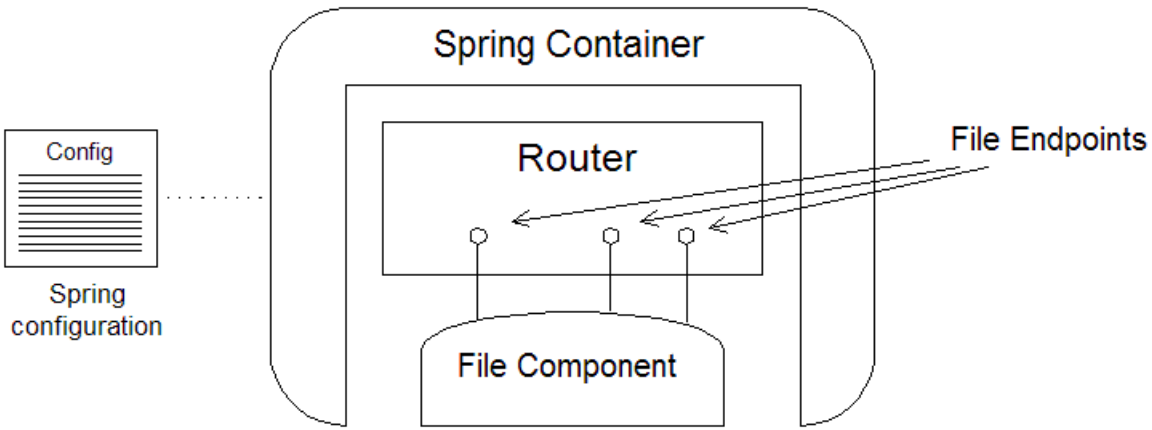
```
machineA:~$ mvn -version
Maven version: 2.0.9
Java version: 1.5.0_16
OS name: "mac os x" version: "10.5.6" arch: "i386" Family:
"unix"
```

Tutorial Overview

Overview

Figure 2.1 on page 20 shows an overview of the architecture of the router featured in this tutorial.

Figure 2.1. Overview of the Tutorial



The router shown in Figure 2.1 on page 20 consists of the following parts:

- **Router** — The core component of the router example. The router consists of an instance of type `org.apache.camel.builder.RouteBuilder`, which defines a route between component endpoints. This route is executed at runtime by the FUSE Mediation Router engine.
- **Spring container** — A standard container (see Spring[1]) that implements sophisticated configuration mechanisms; for example, supporting concepts such as *dependency injection* and *reversion of control*.
- **Spring configuration file** — An XML file that tells the Spring container what objects need to be loaded to start the router. By default, the Spring configuration file is placed in `META-INF/spring/camel-context.xml` on the current classpath.

[1] http://www.springframework.org/

In this example, the Spring container is configured with the name of a Java package, `tutorial`. The Spring container initializes the router artifacts, such as `RouteBuilder` objects, that it finds in the specified Java package.

- **File endpoints** — The source and sinks of the route as specified in the `RouteBuilder` object. In this example, all of the endpoints are *file endpoints*, which are used to read or write messages to the file system.

Tutorial stages

The tutorial consists of the following stages:

1. "Create a New Project"
2. "Examining the Sample Code"
3. "Build and Run the Sample Project"

Create a New Project

Overview

The first step in the tutorial is creating a Maven project (`simple-router`) that contains a sample application.

Steps

To create the project do the following:

1. Create a new directory, *`ProjectRoot`*.
2. In a command window, change to the *`ProjectRoot`* directory.
3. Enter the following command to create the `simple-router` project:

```
mvn archetype:create
-DremoteRepositories=http://repo.open.iona.com/maven2
-DarchetypeGroupId=org.apache.camel.archetypes
-DarchetypeArtifactId=camel-archetype-java
-DarchetypeVersion=1.6.0.0-fuse
-DgroupId=tutorial
-DartifactId=simple-router
```

Note

Maven accesses the Internet to download JARs and stores them in its local repository.

Maven creates the following directories and files:

```
ProjectRoot/simple-router
ProjectRoot/simple-router/pom.xml ❶
ProjectRoot/simple-router/ReadMe.txt
ProjectRoot/simple-router/src
ProjectRoot/simple-router/src/data
ProjectRoot/simple-router/src/data/message1.xml ❷
ProjectRoot/simple-router/src/data/message2.xml ❸
ProjectRoot/simple-router/src/main
ProjectRoot/simple-router/src/main/java
ProjectRoot/simple-router/src/main/java/tutorial
ProjectRoot/simple-router/src/main/java/tutorial/MyRouteBuilder.java ❹
ProjectRoot/simple-router/src/main/resources
ProjectRoot/simple-router/src/main/resources/log4j.properties
ProjectRoot/simple-router/src/main/resources/META-INF
ProjectRoot/simple-router/src/main/resources/META-INF/spring
ProjectRoot/simple-router/src/main/resources/META-INF/spring/camel-context.xml ❺
```

Some of the project artifacts are described below:

❶ `pom.xml` is a Maven project file.

❷ `message1.xml` is an input message in XML format.

❸ `message2.xml` is an input message in XML format.

❹ `MyRouteBuilder.java` is a Java source file that implements the sample route.

❺ `camel-context.xml` is a Spring configuration file.

Examining the Sample Code

Overview

Once the sample project is generated the sample code can be found in

ProjectRoot/simple-router/src/main/java/tutorial/MyRouteBuilder.java. This sample code implements a content-based router, and illustrates how easily — and concisely — you can solve integration problems using FUSE Mediation Router.

Sample Code

Example 2.1 on page 24 shows how the sample route is implemented.

Example 2.1. Sample Content-Based Router

```
...
package tutorial; ❶

import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.spring.Main;

import static org.apache.camel.builder.xml.XPathBuilder.xpath;

/**
* A Camel Router
*
* @version $
*/
public class MyRouteBuilder extends RouteBuilder {

     /**
     * A main() so we can easily run these routing rules in our IDE
     */
     public static void main(String... args) {
          Main.main(args);  ❷
     }

     /**
     * Lets configure the Camel routing rules using Java code...
     */
     public void configure() {

  // TODO create Camel routes here.
```

```
  // here is a sample which processes the input files
  // (leaving them in place - see the 'noop' flag)
  // then performs content based routing on the message
  // using XPath
  from("file:src/data?noop=true"). ❸
      choice().
         when(xpath("/person/city = 'London'")). ❹
             to("file:target/messages/uk").
         otherwise().
             to("file:target/messages/others");

     }
}
```

One of the most notable features of the code in Example 2.1 on page 24 is how the Java DSL uses a series of method calls to create an English-like expression that makes the intent of the code clear — the sample reads input messages from a directory, applies an XPath predicate to each message's XML content, and, based on the result, chooses a different route for the output messages.

Other notable features include:

❶ Maven automatically creates a package name based on the value of the `-DgroupId` argument to the **mvn archetype:create** command.

❷ FUSE Mediation Router defines a convenient wrapper class for the Spring container. To instantiate a Spring container instance, all that is required is a short `main()` method that delegates creation of the container to the wrapper class.

❸ The `from()` method call takes a file URL as its argument. This URL provides information that FUSE Mediation Router uses to interpret and execute the route.

The `file:` prefix in the URL indicates that a file endpoint is desired, which means the file component is responsible for creating the endpoint. The file component, like other FUSE Mediation Router components, serves as an endpoint factory.

The option in the URL, `?noop=true`, indicates that the files in `src/data` should be left in place after they are consumed. This option is one of many available. Like other components, the file component provides numerous options.

❹ The `when()` method call specifies an XPath predicate, which is applied to each input message. If the predicate evaluates to true, the message

is routed to the `uk` subdirectory; if it evaluates to false, the message is routed to the `others` subdirectory.

Build and Run the Sample Project

Overview

Once the route is implemented Maven can build and run the sample project.

Steps

To build and run the sample project do the following:

1. In a command window, change to the *ProjectRoot*/simple-router directory.
2. Enter the following command to build the project:

```
mvn install
```

Maven builds the project and creates a target directory for the build artifacts:

```
ProjectRoot/simple-router/target
ProjectRoot/simple-router/target/simple-router-1.0-SNAPSHOT.jar ❶
ProjectRoot/simple-router/target/classes
ProjectRoot/simple-router/target/classes/log4j.properties ❷
ProjectRoot/simple-router/target/classes/META-INF
ProjectRoot/simple-router/target/classes/META-INF/spring
ProjectRoot/simple-router/target/classes/META-INF/spring/camel-context.xml
ProjectRoot/simple-router/target/classes/tutorial
ProjectRoot/simple-router/target/classes/tutorial/MyRouteBuilder.class ❸
ProjectRoot/simple-router/target/maven-archiver
ProjectRoot/simple-router/target/maven-archiver/pom.properties
```

Some of the project artifacts are described below:

❶ simple-router-1.0-SNAPSHOT.jar is the deployment JAR.

❷ log4j.properties is a properties file used to control logging levels.

❸ MyRouteBuilder.class is the class file compiled from MyRouteBuilder.java.

3. Enter the following command to run the project:

```
mvn camel:run
```

When FUSE Mediation Router starts, it prints lines to the console. For example:

```
23-Feb-2009 16:51:04 org.apache.camel.spring.Main doStart
INFO: Apache Camel 1.6.0.0-fuse starting
23-Feb-2009 16:51:04 org.springframework.context.support.Ab
stractApplicationContext prepareRefresh
...
```

The sample application runs until it is manually stopped. It routes messages from *ProjectRoot*`/simple-router/src/data` to either *ProjectRoot*`/simple-router/target/messages/uk`, or *ProjectRoot*`/simple-router/target/messages/others`.

To stop the application press **Ctrl**+**C**.