Progress FUSE

FUSE™ Mediation Router

Implementing Enterprise Integration Patterns

Version 1.6
April 2009

PROGRESS SOFTWARE

Implementing Enterprise Integration Patterns

Version 1.6

Publication date 17 Jul 2009

Table of Contents

1. Introducing Enterprise Integration Patterns 9
Overview of the Patterns 10
2. Messaging Systems 17
Message 18
Message Channel 21
Message Endpoint 24
Pipes and Filters 26
Message Router 29
Message Translator 31
3. Messaging Channels 33
Point-to-Point Channel 34
Publish-Subscribe Channel 36
Dead Letter Channel 39
Guaranteed Delivery 43
Message Bus 47
4. Message Construction 49
5. Message Routing 51
Content-based Router 52
Message Filter 54
Recipient List 56
Splitter 59
Aggregator 61
Resequencer 66
Routing Slip 70
Throttler 72
Delayer 73
Load Balancer 75
Multicast 79
6. Message Transformation 83
Content Enricher 84
Content Filter 86
Normalizer 88
7. Messaging Endpoints 89
Messaging Mapper 90
Event Driven Consumer 92
Polling Consumer 93
Competing Consumers 95
Message Dispatcher 98
Selective Consumer 101
Durable Subscriber 104
Idempotent Consumer 107

Transactional Client 110
Messaging Gateway 114
Service Activator 115
8. System Management 119
Wire Tap 120
A. Migrating from ServiceMix EIP 121
Migrating Endpoints 122
Common Elements 125
ServiceMix EIP Patterns 127
Content-Based Router 129
Content Enricher 131
Message Filter 133
Pipeline 135
Resequencer 137
Static Recipient List 139
Static Routing Slip 141
Wire Tap 143
XPath Splitter 145

List of Figures

2.1. Message Pattern 18
2.2. Message Channel Pattern 21
2.3. Message Endpoint Pattern 24
2.4. Pipes and Filters Pattern 26
2.5. Pipeline for InOut Exchanges 26
2.6. Pipeline for InOnly Exchanges 27
2.7. Message Router Pattern 29
2.8. Message Translator Pattern 31
3.1. Point to Point Channel Pattern 34
3.2. Publish Subscribe Channel Pattern 36
3.3. Dead Letter Channel Pattern 39
3.4. Guaranteed Delivery Pattern 43
3.5. Message Bus Pattern 47
4.1. Correlation Identifier Pattern 49
5.1. Content-Based Router Pattern 52
5.2. Message Filter Pattern 54
5.3. Recipient List Pattern 56
5.4. Splitter Pattern 59
5.5. Aggregator Pattern 61
5.6. Resequencer Pattern 66
5.7. Routing Slip Pattern 70
5.8. Multicast Pattern 79
6.1. Content Enricher Pattern 84
6.2. Content Filter Pattern 86
6.3. Normalizer Pattern 88
7.1. Event Driven Consumer Pattern 92
7.2. Polling Consumer Pattern 93
7.3. Competing Consumers Pattern 95
7.4. Message Dispatcher Pattern 98
7.5. Selective Consumer Pattern 101
7.6. Durable Subscriber Pattern 105
7.7. Transactional Client Pattern 110
7.8. Messaging Gateway Pattern 114
7.9. Service Activator Pattern 115
8.1. Wire Tap Pattern 120
A.1. Content-Based Router Pattern 129
A.2. Content Enricher Pattern 131
A.3. Message Filter Pattern 133
A.4. Pipes and Filters Pattern 135
A.5. Resequencer Pattern 137
A.6. Static Recipient List Pattern 139

A.7. Wire Tap Pattern 143
A.8. XPath Splitter Pattern 145

List of Tables

1.1. Messaging Systems 10
1.2. Messaging Channels 11
1.3. Message Construction 12
1.4. Message Routing 12
1.5. Message Transformation 13
1.6. Messaging Endpoints 14
1.7. System Management 15
3.1. Redelivery Policy Settings 40
3.2. Dead Letter Redelivery Headers 41
5.1. Splitter Headers 59
A.1. Mapping the Exchange Target Element 125
A.2. ServiceMix EIP Patterns 127

Chapter 1. Introducing Enterprise Integration Patterns

The FUSE Mediation Router's Enterprise Integration Patterns are inspired by a book of the same name written by Gregor Hohpe and Bobby Woolf. The patterns described by these authors provide an excellent toolbox for developing enterprise integration projects. In addition to providing a common language for discussing integration architectures, many of the patterns can be implemented directly using FUSE Mediation Router's programming interfaces and XML configuration.

Overview of the Patterns 10

Overview of the Patterns

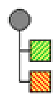
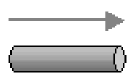
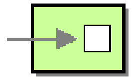
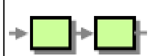
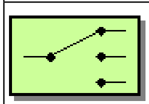
Enterprise Integration Patterns book

FUSE Mediation Router supports most of the patterns from the book, Enterprise Integration Patterns[1] by Gregor Hohpe and Bobby Woolf.

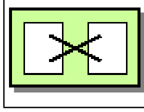
Messaging systems

The messaging systems patterns, shown in Table 1.1 on page 10, introduce the fundamental concepts and components that make up a messaging system.

Table 1.1. Messaging Systems

Icon	Name	Use Case
	Message	How can two applications connected by a message channel exchange a piece of information?
	Message Channel	How does one application communicate with another application using messaging?
	Message Endpoint	How does an application connect to a messaging channel to send and receive messages?
	Pipes and Filters	How can we perform complex processing on a message while still maintaining independence and flexibility?
	Message Router	How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of defined conditions?

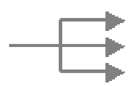
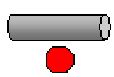
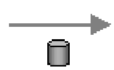
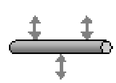
[1] http://www.enterpriseintegrationpatterns.com/toc.html

Icon	Name	Use Case
	Message Translator	How do systems using different data formats communicate with each other using messaging?

Messaging channels

A messaging channel is the basic component used for connecting the participants in a messaging system. The patterns in Table 1.2 on page 11 describe the different kinds of messaging channels available.

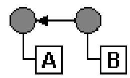
Table 1.2. Messaging Channels

Icon	Name	Use Case
	Point to Point Channel	How can the caller be sure that exactly one receiver will receive the document or will perform the call?
	Publish Subscribe Channel	How can the sender broadcast an event to all interested receivers?
	Dead Letter Channel	What will the messaging system do with a message it cannot deliver?
	Guaranteed Delivery	How does the sender make sure that a message will be delivered, even if the messaging system fails?
	Message Bus	What is an architecture that enables separate, decoupled applications to work together, such that one or more of the applications can be added or removed without affecting the others?

Message construction

The message construction patterns, shown in Table 1.3 on page 12, describe the various forms and functions of the messages that pass through the system.

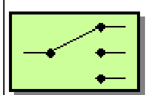
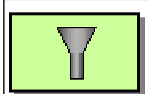
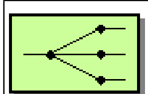
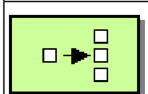
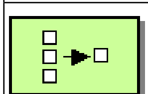
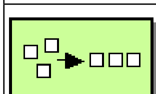
Table 1.3. Message Construction

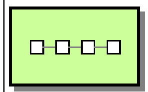
Icon	Name	Use Case
	Correlation Identifier	How does a requestor identify the request that generated the received reply?

Message routing

The message routing patterns, shown in Table 1.4 on page 12, describe various ways of linking message channels together, including various algorithms that can be applied to the message stream (without modifying the body of the message).

Table 1.4. Message Routing

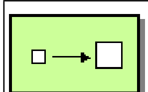
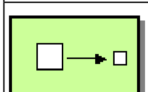
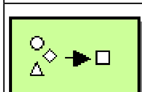
Icon	Name	Use Case
	Content Based Router	How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?
	Message Filter	How does a component avoid receiving uninteresting messages?
	Recipient List	How do we route a message to a list of dynamically specified recipients?
	Splitter	How can we process a message if it contains multiple elements, each of which might have to be processed in a different way?
	Aggregator	How do we combine the results of individual, but related messages so that they can be processed as a whole?
	Resequencer	How can we get a stream of related, but out-of-sequence, messages back into the correct order?

Icon	Name	Use Case
	Routing Slip	How do we route a message consecutively through a series of processing steps when the sequence of steps is not known at design-time, and might vary for each message?
	Throttler	How can I throttle messages to ensure that a specific endpoint does not get overloaded, or that we don't exceed an agreed SLA with some external service?
	Delayer	How can I delay the sending of a message?
	Load Balancer	How can I balance load across a number of endpoints?
	Multicast	How can I route a message to a number of endpoints at the same time?

Message transformation

The message transformation patterns, shown in Table 1.5 on page 13, describe how to modify the contents of messages for various purposes.

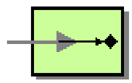
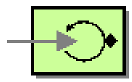
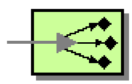
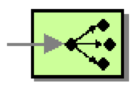
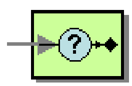
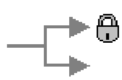
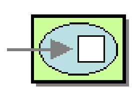
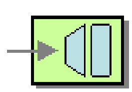
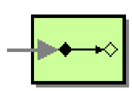
Table 1.5. Message Transformation

Icon	Name	Use Case
	Content Enricher	How do we communicate with another system if the message originator does not have all the required data items available?
	Content Filter	How do you simplify dealing with a large message, when you are interested in only a few data items?
	Normalizer	How do you process messages that are semantically equivalent, but arrive in a different format?

Messaging endpoints

A messaging endpoint denotes the point of contact between a messaging channel and an application. The messaging endpoint patterns, shown in Table 1.6 on page 14, describe various features and qualities of service that can be configured on an endpoint.

Table 1.6. Messaging Endpoints

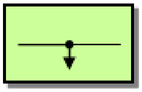
Icon	Name	Use Case
	Messaging Mapper on page 90	How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?
	Event Driven Consumer	How can an application automatically consume messages as they become available?
	Polling Consumer	How can an application consume a message when the application is ready?
	Competing Consumers on page 95	How can a messaging client process multiple messages concurrently?
	Message Dispatcher	How can multiple consumers on a single channel coordinate their message processing?
	Selective Consumer	How can a message consumer select which messages it wants to receive?
	Durable Subscriber	How can a subscriber avoid missing messages when it's not listening for them?
	Idempotent Consumer	How can a message receiver deal with duplicate messages?
	Transactional Client	How can a client control its transactions with the messaging system?
	Messaging Gateway	How do you encapsulate access to the messaging system from the rest of the application?
	Service Activator	How can an application design a service to be invoked both via various

Icon	Name	Use Case
		messaging technologies and via non-messaging techniques?

System management

The system management patterns, shown in Table 1.7 on page 15, describe how to monitor, test, and administer a messaging system.

Table 1.7. System Management

Icon	Name	Use Case
	Wire Tap	How do you inspect messages that travel on a point-to-point channel?

Chapter 2. Messaging Systems

This chapter introduces the fundamental building blocks of a messaging system, such as endpoints, messaging channels, and message routers.

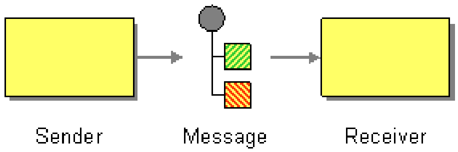
Message 18
Message Channel 21
Message Endpoint 24
Pipes and Filters 26
Message Router 29
Message Translator 31

Message

Overview

A *message* is the smallest unit for transmitting data in a messaging system (represented by the grey dot in the figure below). The message itself might have some internal structure—for example, a message containing multiple parts—which is represented by geometrical figures attached to the grey dot in Figure 2.1 on page 18.

Figure 2.1. Message Pattern



Types of message

FUSE Mediation Router defines the following distinct message types:

- *In* message — A message that travels through a route from a consumer endpoint to a producer endpoint (typically, initiating a message exchange).
- *Out* message — A message that travels through a route from a producer endpoint back to a consumer endpoint (usually, in response to an *In* message).
- *Fault* message *(deprecated)* — A message that travels through a route from a producer endpoint back to a consumer endpoint, for the purpose of indicating an exception or an error condition (usually in response to an *In* message).

Note

The fault message type is deprecated and will be replaced by a different mechanism in a future release of FUSE Mediation Router.

All of these message types are represented internally by the `org.apache.camel.Message` interface.

Message structure

By default, FUSE Mediation Router applies the following structure to all message types:

- *Headers* — Contains metadata or header data extracted from the message.
- *Body* — Usually contains the entire message in its original form.
- *Attachments* — Message attachments (required for integrating with certain messaging systems, such as JBI[1]).

It is important to remember that this division into headers, body, and attachments is an abstract model of the message. FUSE Mediation Router supports many different components, that generate a wide variety of message formats. Ultimately, it is the underlying component implementation that decides what gets placed into the headers and body of a message.

Correlating messages

Internally, FUSE Mediation Router remembers the message IDs, which are used to correlate individual messages. In practice, however, the most important way that FUSE Mediation Router correlates messages is through *exchange* objects.

Exchange objects

An exchange object is an entity that encapsulates related messages, where the collection of related messages is referred to as a *message exchange* and the rules governing the sequence of messages are referred to as an *exchange pattern*. For example, two common exchange patterns are: one-way event messages (consisting of an *In* message), and request-reply exchanges (consisting of an *In* message, followed by an *Out* message).

Accessing messages

When defining a routing rule in the Java DSL, you can access the headers and body of a message using the following DSL builder methods:

- `header(String name)`, `body()` — Returns the named header and the body of the current *In* message.
- `outBody()` — Returns the body of the current *Out* message.

[1] http://java.sun.com/integration/

For example, to populate the *In* message's `username` header, you can use the following Java DSL route:

```
from(SourceURL).setHeader("username", "John.Doe").to(TargetURL);
```

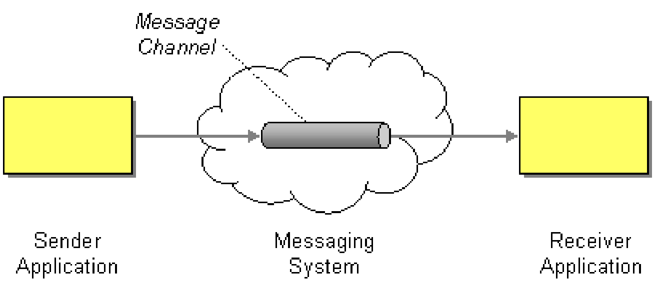
Message Channel

Overview

A *message channel* is a logical channel in a messaging system. That is, sending messages to different message channels provides an elementary way of sorting messages into different message types. Message queues and message topics are examples of message channels. You should remember that a logical channel is *not* the same as a physical channel. There can be several different ways of physically realizing a logical channel.

In FUSE Mediation Router, a message channel is represented by an endpoint URI of a message-oriented component as shown in Figure 2.2 on page 21.

Figure 2.2. Message Channel Pattern



Message-oriented components

The following message-oriented components in FUSE Mediation Router support the notion of a message channel:

- "ActiveMQ"
- "JMS"
- "MSMQ"
- "AMQP"

ActiveMQ

In ActiveMQ, message channels are represented by *queues* or *topics*. The endpoint URI for a specific queue, *QueueName*, has the following format:

```
activemq:QueueName
```

The endpoint URI for a specific topic, *TopicName*, has the following format:

```
activemq:topic:TopicName
```

For example, to send messages to the queue, `Foo.Bar`, use the following endpoint URI:

```
activemq:Foo.Bar
```

See ???? for more details and instructions on setting up the ActiveMQ component.

JMS

The Java Messaging Service (JMS) is a generic wrapper layer that is used to access many different kinds of message systems (for example, you can use it to wrap ActiveMQ, MQSeries, Tibco, BEA, Sonic, and others). In JMS, message channels are represented by queues, or topics. The endpoint URI for a specific queue, *QueueName*, has the following format:

```
jms:QueueName
```

The endpoint URI for a specific topic, *TopicName*, has the following format:

```
jms:topic:TopicName
```

See ???? for more details and instructions on setting up the JMS component.

MSMQ

The Microsoft Message Queuing (MSMQ) technology is a queuing system that runs on Windows Server machines (see Microsoft Message Queuing[2]). In MSMQ, you can access queues using an endpoint URI with the following format:

```
msmq:MSMQueueName
```

Where the *MSMQueueName* is a queue reference, defined according to the rules of MSMQ. You can reference a queue using any of the approaches described in Referencing a Queue[3].

See ???? for more details.

AMQP

In AMQP, message channels are represented by queues, or topics. The endpoint URI for a specific queue, *QueueName*, has the following format:

[2] http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx#ECC
[3] http://msdn2.microsoft.com/en-us/library/ms704998%28VS.85%29.aspx

```
amqp:QueueName
```

The endpoint URI for a specific topic, *TopicName*, has the following format:

```
amqp:topic:TopicName
```

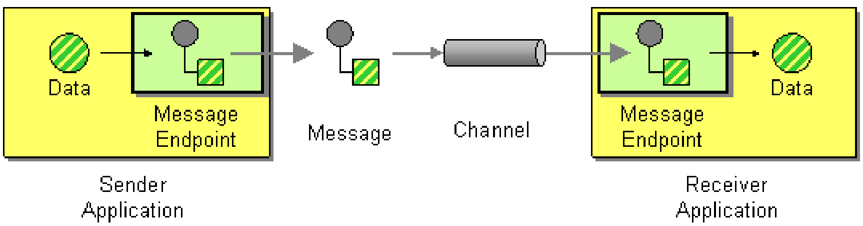
See ???? for more details and instructions on setting up the AMQP component.

Message Endpoint

Overview

A *message endpoint* is the interface between an application and a messaging system. As shown in Figure 2.3 on page 24, you can have a sender endpoint, sometimes called a proxy or a service consumer, which is responsible for sending *In* messages, and a receiver endpoint, sometimes called an endpoint or a service, which is responsible for receiving *In* messages.

Figure 2.3. Message Endpoint Pattern



Types of endpoint

FUSE Mediation Router defines two basic types of endpoint:

- *Consumer endpoint* — Appears at the start of a FUSE Mediation Router route and reads *In* messages from an incoming channel (equivalent to a *receiver* endpoint).
- *Producer endpoint* — Appears at the end of a FUSE Mediation Router route and writes *In* messages to an outgoing channel (equivalent to a *sender* endpoint). It is possible to define a route with multiple producer endpoints.

Endpoint URIs

In FUSE Mediation Router, an endpoint is represented by an *endpoint URI*, which typically encapsulates the following kinds of data:

- *Endpoint URI for a consumer endpoint* — Advertises a specific location (for example, to expose a service to which senders can connect). Alternatively, the URI can specify a message source, such as a message queue. The endpoint URI can include settings to configure the endpoint.
- *Endpoint URI for a producer endpoint* — Contains details of where to send messages and includes the settings to configure the endpoint. In some

cases, the URI specifies the location of a remote receiver endpoint; in other cases, the destination can have an abstract form, such as a queue name.

An endpoint URI in FUSE Mediation Router has the following general form:

```
ComponentPrefix:ComponentSpecificURI
```

Where *`ComponentPrefix`* is a URI prefix that identifies a particular FUSE Mediation Router component (see ???? for details of all the supported components). The remaining part of the URI, *`ComponentSpecificURI`*, has a syntax defined by the particular component. For example, to connect to the JMS queue, `Foo.Bar`, you can define an endpoint URI like the following:

```
jms:Foo.Bar
```

To define a route that connects the consumer endpoint, `file://local/router/messages/foo`, directly to the producer endpoint, `jms:Foo.Bar`, you can use the following Java DSL fragment:

```
from("file://local/router/messages/foo").to("jms:Foo.Bar");
```

Alternatively, you can define the same route in XML, as follows:

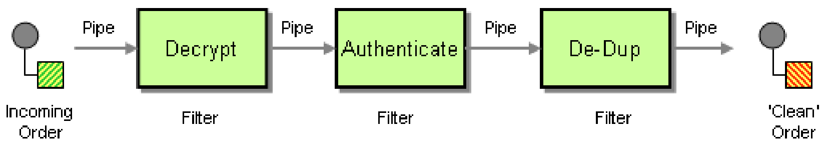
```
<camelContext id="CamelContextID" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="file://local/router/messages/foo"/>
    <to uri="jms:Foo.Bar"/>
  </route>
</camelContext>
```

Pipes and Filters

Overview

The *pipes and filters* pattern, shown in Figure 2.4 on page 26, describes a way of constructing a route by creating a chain of filters, where the output of one filter is fed into the input of the next filter in the pipeline (analogous to the UNIX `pipe` command). The advantage of the pipeline approach is that it enables you to compose services (some of which can be external to the FUSE Mediation Router application) to create more complex forms of message processing.

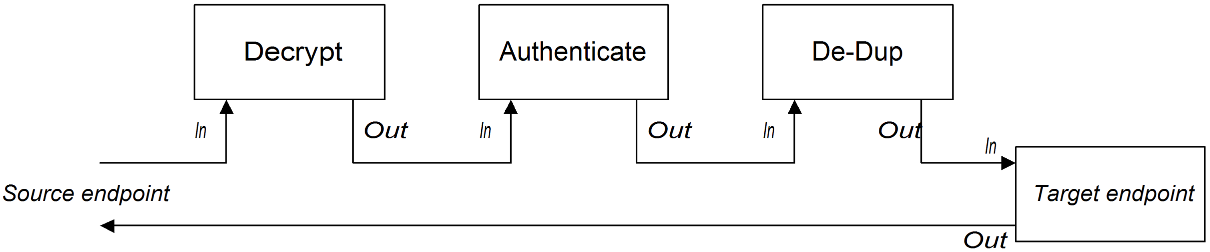
Figure 2.4. Pipes and Filters Pattern



Pipeline for the InOut exchange pattern

Normally, all of the endpoints in a pipeline have an input (*In* message) and an output (*Out* message), which implies that they are compatible with the *InOut* message exchange pattern. A typical message flow through an *InOut* pipeline is shown in Figure 2.5 on page 26.

Figure 2.5. Pipeline for InOut Exchanges



Where the pipeline connects the output of each endpoint to the input of the next one. The *Out* message from the final endpoint gets sent back to the original caller. You can define a route for this pipeline, as follows:

```
from("jms:RawOrders").pipeline("cxf:bean:decrypt",
"cxf:bean:authenticate", "cxf:bean:dedup", "jms:CleanOrders");
```

The same route can be configured in XML, as follows:

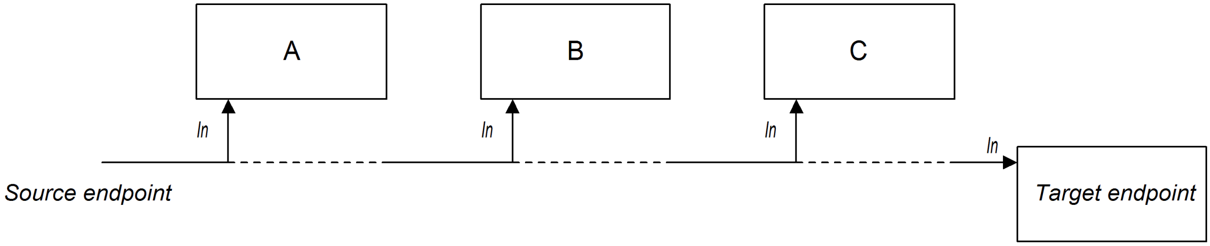
```
<camelContext id="buildPipeline" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="jms:RawOrders"/>
    <to uri="cxf:bean:decrypt"/>
    <to uri="cxf:bean:authenticate"/>
    <to uri="cxf:bean:dedup"/>
    <to uri="jms:CleanOrders"/>
  </route>
</camelContext>
```

There is no dedicated pipeline element in XML. The preceding combination of `from` and `to` elements is semantically equivalent to a pipeline. See "Comparison of pipeline() and to() DSL commands" on page 27.

Pipeline for the InOnly and RobustInOnly exchange patterns

When there are no *Out* messages available from the endpoints in the pipeline (as is the case for the `InOnly` and `RobustInOnly` exchange patterns), a pipeline cannot be connected in the normal way. In this special case, the pipeline is constructed by passing a copy of the original *In* message to each of the endpoints in the pipeline, as shown in Figure 2.6 on page 27. This type of pipeline is equivalent to a recipient list with fixed destinations(see "Recipient List" on page 56).

Figure 2.6. Pipeline for InOnly Exchanges



The route for this pipeline is defined using the same syntax as an *InOut* pipeline (either in Java DSL or in XML).

Comparison of pipeline() and to() DSL commands

In the Java DSL, you can define a pipeline route using either of the following syntaxes:

- *Using the pipeline() processor command* — Use the pipeline processor to construct a pipeline route as follows:

```
from(SourceURI).pipeline(FilterA, FilterB, TargetURI);
```

- *Using the to() command* — Use the `to()` command to construct a pipeline route as follows:

  ```
  from(SourceURI).to(FilterA, FilterB, TargetURI);
  ```

  Alternatively, you can use the equivalent syntax:

  ```
  from(SourceURI).to(FilterA).to(FilterB).to(TargetURI);
  ```

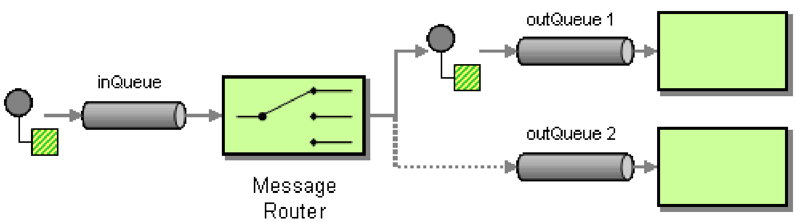
Exercise caution when using the `to()` command syntax, because it is *not* always equivalent to a pipeline processor. In Java DSL, the meaning of `to()` can be modified by the preceding command in the route. For example, when the `multicast()` command precedes the `to()` command, it binds the listed endpoints into a multicast pattern, instead of a pipeline pattern(see "Multicast" on page 79).

Message Router

Overview

A *message router*, shown in Figure 2.7 on page 29, is a type of filter that consumes messages from a single consumer endpoint and redirects them to the appropriate target endpoint, based on a particular decision criterion. A message router is concerned only with redirecting messages; it does not modify the message content.

Figure 2.7. Message Router Pattern



A message router can easily be implemented in FUSE Mediation Router using the `choice()` processor, where each of the alternative target endpoints can be selected using a `when()` subclause (for details of the choice processor, see "Processors" in *Defining Routes* and "Processors" in *Defining Routes*)

Java DSL example

The following Java DSL example shows how to route messages to three alternative destinations (either `seda:a`, `seda:b`, or `seda:c`) depending on the contents of the `foo` header:

```
from("seda:a").choice()
    .when(header("foo").isEqualTo("bar")).to("seda:b")
    .when(header("foo").isEqualTo("cheese")).to("seda:c")
    .otherwise().to("seda:d");
```

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext id="buildSimpleRouteWithChoice" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <choice>
      <when>
        <xpath>$foo = 'bar'</xpath>
```

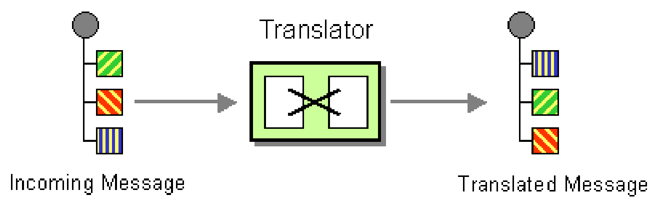
```
        <to uri="seda:b"/>
      </when>
      <when>
        <xpath>$foo = 'cheese'</xpath>
        <to uri="seda:c"/>
      </when>
      <otherwise>
        <to uri="seda:d"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

Message Translator

Overview

The *message translator* pattern, shown in Figure 2.8 on page 31 describes a component that modifies the contents of a message, translating it to a different format. You can use FUSE Mediation Router's bean integration feature to perform the message translation.

Figure 2.8. Message Translator Pattern



Bean integration

You can transform a message using bean integration, which enables you to call a method on any registered bean. For example, to call the method, `myMethodName()`, on the bean with ID, `myTransformerBean`:

```
from("activemq:SomeQueue")
  .beanRef("myTransformerBean", "myMethodName")
  .to("mqseries:AnotherQueue");
```

Where the `myTransformerBean` bean is defined in either a Spring XML file or in JNDI. If, you omit the method name parameter from `beanRef()`, the bean integration will try to deduce the method name to invoke by examining the message exchange.

You can also add your own explicit `Processor` instance to perform the transformation, as follows:

```
from("direct:start").process(new Processor() {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
    }
}).to("mock:result");
```

Or, you can use the DSL to explicitly configure the transformation, as follows:

```
from("direct:start").setBody(body().append("
World!")).to("mock:result");
```

You can also use *templating* to consume a message from one destination, transform it with something like Velocity or XQuery and then send it on to another destination. For example, using the *InOnly* exchange pattern (one-way messaging) :

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

If you want to use *InOut* (request-reply) semantics to process requests on the `My.Queue` queue on ActiveMQ with a template generated response, then you could use a route like the following to send responses back to the `JMSReplyTo` destination:

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

Chapter 3. Messaging Channels

Messaging channels provide the plumbing for a messaging application. This chapter describes the different kinds of messaging channels available in a messaging system, and the roles that they play.

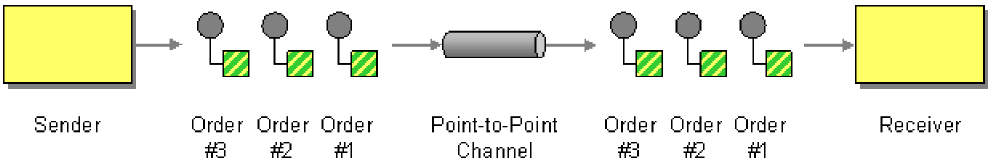
Point-to-Point Channel 34
Publish-Subscribe Channel 36
Dead Letter Channel 39
Guaranteed Delivery 43
Message Bus 47

Point-to-Point Channel

Overview

A *point-to-point channel*, shown in Figure 3.1 on page 34 is a message channel on page 21 that guarantees that only one receiver consumes any given message. This is in contrast with a publish-subscribe channel on page 36, which allows multiple receivers to consume the same message. In particular, with a point-to-point channel, it is possible for multiple receivers to subscribe to the same channel. If more than one receiver competes to consume a message, it is up to the message channel to ensure that only one receiver actually consumes the message.

Figure 3.1. Point to Point Channel Pattern



Components that support point-to-point channel

The following FUSE Mediation Router components support the point-to-point channel pattern:

- "JMS"
- "ActiveMQ"
- "SEDA"
- "JPA"
- "XMPP"

JMS

In JMS, a point-to-point channel is represented by a *queue*. For example, you can specify the endpoint URI for a JMS queue called `Foo.Bar` as follows:

```
jms:queue:Foo.Bar
```

The qualifier, `queue:`, is optional, because the JMS component creates a queue endpoint by default. Therefore, you can also specify the following equivalent endpoint URI:

```
jms:Foo.Bar
```

See ???? for more details.

ActiveMQ

In ActiveMQ, a point-to-point channel is represented by a queue. For example, you can specify the endpoint URI for an ActiveMQ queue called `Foo.Bar` as follows:

```
activemq:queue:Foo.Bar
```

See ???? for more details.

SEDA

The FUSE Mediation Router Staged Event-Driven Architecture (SEDA) component is implemented using a blocking queue. Use the SEDA component if you want to create a lightweight point-to-point channel that is *internal* to the FUSE Mediation Router application. For example, you can specify an endpoint URI for a SEDA queue called `SedaQueue` as follows:

```
seda:SedaQueue
```

JPA

The Java Persistence API (JPA) component is an EJB 3 persistence standard that is used to write entity beans out to a database. See ???? for more details.

XMPP

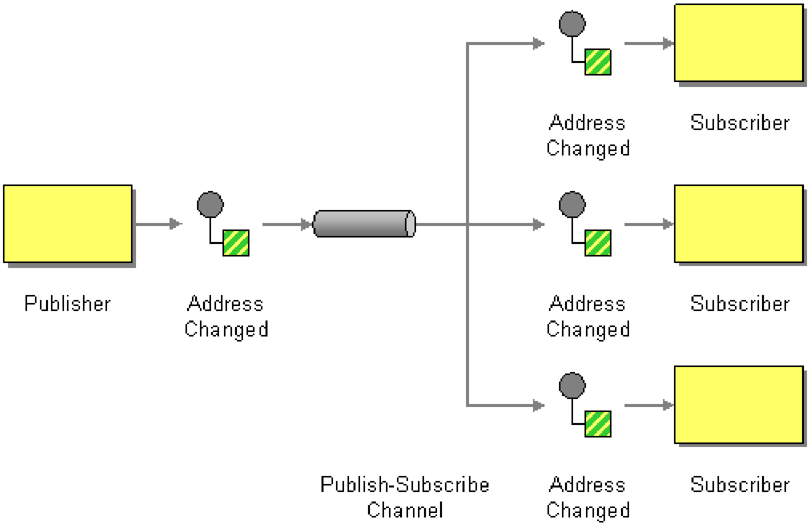
The XMPP (Jabber) component supports the point-to-point channel pattern when it is used in the person-to-person mode of communication. See ???? for more details.

Publish-Subscribe Channel

Overview

A *publish-subscribe channel*, shown in Figure 3.2 on page 36, is a message channel on page 21 that enables multiple subscribers to consume any given message. This is in contrast with a point-to-point channel on page 34. Publish-subscribe channels are frequently used as a means of broadcasting events or notifications to multiple subscribers.

Figure 3.2. Publish Subscribe Channel Pattern



Components that support publish-subscribe channel

The following FUSE Mediation Router components support the publish-subscribe channel pattern:

- "JMS"
- "ActiveMQ"

- "XMPP"

JMS

In JMS, a publish-subscribe channel is represented by a *topic*. For example, you can specify the endpoint URI for a JMS topic called `StockQuotes` as follows:

```
jms:topic:StockQuotes
```

See ???? for more details.

ActiveMQ

In ActiveMQ, a publish-subscribe channel is represented by a topic. For example, you can specify the endpoint URI for an ActiveMQ topic called `StockQuotes`, as follows:

```
activemq:topic:StockQuotes
```

See ???? for more details.

XMPP

The XMPP (Jabber) component supports the publish-subscribe channel pattern when it is used in the group communication mode. See ???? for more details.

Static subscription lists

If you prefer, you can also implement publish-subscribe logic within the FUSE Mediation Router application itself. A simple approach is to define a *static subscription list*, where the target endpoints are all explicitly listed at the end of the route. However, this approach is not as flexible as a JMS or ActiveMQ topic.

Java DSL example

The following Java DSL example shows how to simulate a publish-subscribe channel with a single publisher, `seda:a`, and three subscribers, `seda:b`, `seda:c`, and `seda:d`:

```
from("seda:a").to("seda:b", "seda:c", "seda:d");
```

Note

This only works for the *InOnly* message exchange pattern.

XML configuration example

The following example shows how to configure the same route in XML:

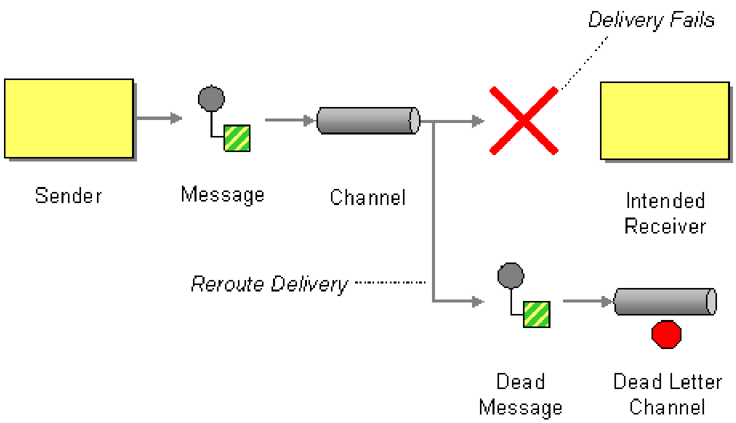
```
<camelContext id="buildStaticRecipientList" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:b"/>
    <to uri="seda:c"/>
    <to uri="seda:d"/>
  </route>
</camelContext>
```

Dead Letter Channel

Overview

The *dead letter channel* pattern, shown in Figure 3.3 on page 39, describes the actions to take when the messaging system fails to deliver a message to the intended recipient. This includes such features as retrying delivery and, if delivery ultimately fails, sending the message to a dead letter channel, which archives the undelivered messages.

Figure 3.3. Dead Letter Channel Pattern



Creating a dead letter channel in Java DSL

The following example shows how to create a dead letter channel using Java DSL:

```
errorHandler(deadLetterChannel("seda:errors"));
from("seda:a").to("seda:b");
```

Where the `errorHandler()` method is a Java DSL interceptor, which implies that *all* of the routes defined in the current route builder are affected by this setting. The `deadLetterChannel()` method is a Java DSL command that creates a new dead letter channel with the specified destination endpoint, `seda:errors`.

The `errorHandler()` interceptor provides a catch-all mechanism for handling *all* error types. If you want to apply a more fine-grained approach to exception handling, you can use the `onException` clauses instead(see "onException clause" on page 41).

Redelivery policy

Normally, you do not send a message straight to the dead letter channel, if a delivery attempt fails. Instead, you re-attempt delivery up to some maximum limit, and after all redelivery attempts fail you would send the message to the dead letter channel. To customize message redelivery, you can configure the dead letter channel to have a *redelivery policy*. For example, to specify a maximum of two redelivery attempts, and to apply an exponential backoff algorithm to the time delay between delivery attempts, you can configure the dead letter channel as follows:

```
errorHandler(deadLetterChannel("seda:errors").maximumRedeliv
eries(2).useExponentialBackOff());
from("seda:a").to("seda:b");
```

Where you set the redelivery options on the dead letter channel by invoking the relevant methods in a chain (each method in the chain returns a reference to the current `RedeliveryPolicy` object). Table 3.1 on page 40 summarizes the methods that you can use to set redelivery policies.

Table 3.1. Redelivery Policy Settings

Method Signature	Default	Description
`backOffMultiplier(double multiplier)`	2	If exponential backoff is enabled, let `m` be the backoff multiplier and let `d` be the initial delay. The sequence of redelivery attempts are then timed as follows: `d, m*d, m*m*d, m*m*m*d, ...`
`collisionAvoidancePercent(double collisionAvoidancePercent)`	15	If collision avoidance is enabled, let `p` be the collision avoidance percent. The collision avoidance policy then tweaks the next delay by a random amount, up to plus/minus `p%` of its current value.
`initialRedeliveryDelay(long initialRedeliveryDelay)`	1000	Specifies the delay (in milliseconds) before attempting the first redelivery.
`maximumRedeliveries(int maximumRedeliveries)`	6	Maximum number of delivery attempts.
`useCollisionAvoidance()`	false	Enables collision avoidence, which adds some randomization to the backoff timings to reduce contention probability.

Method Signature	Default	Description
`useExponentialBackOff()`	`false`	Enables exponential backoff.

Redelivery headers

If FUSE Mediation Router attempts to redeliver a message, it automatically sets the headers described in Table 3.2 on page 41 on the *In* message.

Table 3.2. Dead Letter Redelivery Headers

Header Name	Type	Description
`org.apache.camel.RedeliveryCounter`	`Integer`	Counts the number of unsuccessful delivery attempts.
`org.apache.camel.Redelivered`	`Boolean`	True, if one or more redelivery attempts have been made.

onException clause

Instead of using the `errorHandler()` interceptor in your route builder, you can define a series of `onException()` clauses that define different redelivery policies and different dead letter channels for various exception types. For example, to define distinct behavior for each of the `NullPointerException`, `IOException`, and `Exception` types, you can define the following rules in your route builder using Java DSL:

```
onException(NullPointerException.class)
    .maximumRedeliveries(1)
    .setHeader("messageInfo", "Oh dear! An NPE.")
    .to("mock:npe_error");

onException(IOException.class)
    .initialRedeliveryDelay(5000L)
    .maximumRedeliveries(3)
    .backOffMultiplier(1.0)
    .useExponentialBackOff()
    .setHeader("messageInfo", "Oh dear! Some kind of I/O ex
ception.")
    .to("mock:io_error");

onException(Exception.class)
    .initialRedeliveryDelay(1000L)
    .maximumRedeliveries(2)
    .setHeader("messageInfo", "Oh dear! An exception.")
    .to("mock:error");

from("seda:a").to("seda:b");
```

Where the redelivery options are specified by chaining the redelivery policy methods (as listed in Table 3.1 on page 40), and you specify the dead letter channel's endpoint using the `to()` DSL command. You can also call other Java DSL commands in the `onException()` clauses. For example, the preceding example calls `setHeader()` to record some error details in a message header named, `messageInfo`.

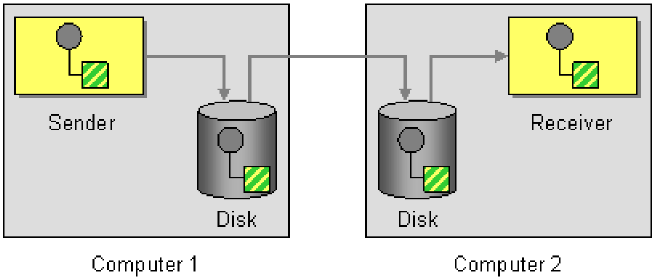
In this example, the `NullPointerException` and the `IOException` exception types are configured specially. All other exception types are handled by the generic `Exception` exception interceptor. By default, FUSE Mediation Router applies the exception interceptor that most closely matches the thrown exception. If it fails to find an exact match, it tries to match the closest base type, and so on. Finally, if no other interceptor matches, the interceptor for the `Exception` type matches all remaining exceptions.

Guaranteed Delivery

Overview

Guaranteed delivery means that once a message is placed into a message channel, the messaging system guarantees that the message will reach its destination, even if parts of the application should fail. In general, messaging systems implement the guaranteed delivery pattern, shown in Figure 3.4 on page 43, by writing messages to persistent storage before attempting to deliver them to their destination.

Figure 3.4. Guaranteed Delivery Pattern



Components that support guaranteed delivery

The following FUSE Mediation Router components support the guaranteed delivery pattern:

- "JMS"
- "ActiveMQ"
- "ActiveMQ Journal"

JMS

In JMS, the `deliveryPersistent` query option indicates whether or not persistent storage of messages is enabled. Usually it is unnecessary to set this option, because the default behavior is to enable persistent delivery. To configure all the details of guaranteed delivery, it is necessary to set configuration options on the JMS provider. These details vary, depending on what JMS provider you are using. For example, MQSeries, TibCo, BEA, Sonic, and others, all provide various qualities of service to support guaranteed delivery.

See ???? for more details.

ActiveMQ

In ActiveMQ, message persistence is enabled by default. From version 5 onwards, ActiveMQ uses the AMQ message store as the default persistence mechanism. There are several different approaches you can use to enabe message persistence in ActiveMQ.

The simplest option (different from Figure 3.4 on page 43) is to enable persistence in a central broker and then connect to that broker using a reliable protocol. After a message is been sent to the central broker, delivery to consumers is guaranteed. For example, in the FUSE Mediation Router configuration file, `META-INF/spring/camel-context.xml`, you can configure the ActiveMQ component to connect to the central broker using the OpenWire/TCP protocol as follows:

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.compon
ent.ActiveMQComponent">
    <property name="brokerURL" value="tcp://somehost:61616"/>

  </bean>
  ...
</beans>
```

If you prefer to implement an architecture where messages are stored locally before being sent to a remote endpoint (similar to Figure 3.4 on page 43), you do this by instantiating an embedded broker in your FUSE Mediation Router application. A simple way to achieve this is to use the ActiveMQ Peer-to-Peer protocol, which implicitly creates an embedded broker to communicate with other peer endpoints. For example, in the `camel-context.xml` configuration file, you can configure the ActiveMQ component to connect to all of the peers in group, `GroupA`, as follows:

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.compon
ent.ActiveMQComponent">
    <property name="brokerURL" value="peer://GroupA/broker1"/>

  </bean>
  ...
</beans>
```

Where `broker1` is the broker name of the embedded broker (other peers in the group should use different broker names). One limiting feature of the

Peer-to-Peer protocol is that it relies on IP multicast to locate the other peers in its group. This makes it unsuitable for use in wide area networks (and in some local area networks that do not have IP multicast enabled).

A more flexible way to create an embedded broker in the ActiveMQ component is to exploit ActiveMQ's VM protocol, which connects to an embedded broker instance. If a broker of the required name does not already exist, the VM protocol automatically creates one. You can use this mechanism to create an embedded broker with custom configuration. For example:

```
<beans ... >
  ...
  <bean id="activemq" class="org.apache.activemq.camel.compon
ent.ActiveMQComponent">
    <property name="brokerURL" value="vm://broker1?brokerCon
fig=xbean:activemq.xml"/>
  </bean>
  ...
</beans>
```

Where `activemq.xml` is an ActiveMQ file which configures the embedded broker instance. Within the ActiveMQ configuration file, you can choose to enable one of the following persistence mechanisms:

- *AMQ persistence(the default)* — A fast and reliable message store that is native to ActiveMQ. For details, see amqPersistenceAdapter[1] and AMQ Message Store[2].
- *JDBC persistence* — Uses JDBC to store messages in any JDBC-compatible database. For details, see jdbcPersistenceAdapter[3] and ActiveMQ Persistence[4].
- *Journal persistence* — A fast persistence mechanism that stores messages in a rolling log file. For details, see journalPersistenceAdapter[5] and ActiveMQ Persistence[6].

[1] http://tinyurl.com/activemq-amqPersistenceAdapter
[2] http://activemq.apache.org/amq-message-store.html
[3] http://tinyurl.com/activemq-jdbPersistenceAdapter
[4] http://activemq.apache.org/persistence.html
[5] http://tinyurl.com/activemq-journalPA
[6] http://activemq.apache.org/persistence.html

- *Kaha persistence* — A persistence mechanism developed specifically for ActiveMQ. For details, see kahaPersistenceAdapter[7] and ActiveMQ Persistence[8].

See ???? for more details.

ActiveMQ Journal

The ActiveMQ Journal component is optimized for a special use case where multiple, concurrent producers write messages to queues, but there is only one active consumer. Messages are stored in rolling log files and concurrent writes are aggregated to boost efficiency.

See ???? for more details.

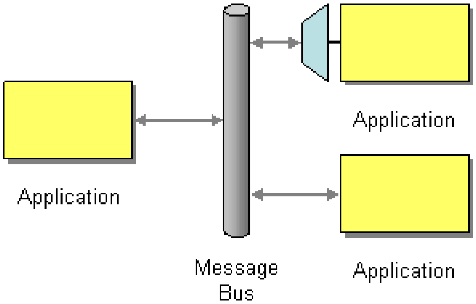
[7] http://tinyurl.com/activemq-kahaPA
[8] http://activemq.apache.org/persistence.html

Message Bus

Overview

Message bus refers to a messaging architecture, shown in Figure 3.5 on page 47, that enables you to connect diverse applications running on diverse computing platforms. In effect, the FUSE Mediation Router and its components constitute a message bus.

Figure 3.5. Message Bus Pattern



The following features of the message bus pattern are reflected in FUSE Mediation Router:

- *Common communication infrastructure* — The router itself provides the core of the common communication infrastructure in FUSE Mediation Router. However, in contrast to some message bus architectures, FUSE Mediation Router provides a heterogeneous infrastructure: messages can be sent into the bus using a wide variety of different transports and using a wide variety of different message formats.
- *Adapters* — Where necessary, the FUSE Mediation Router can translate message formats and propagate messages using different transports. In effect, the FUSE Mediation Router is capable of behaving like an adapter, so that external applications can hook into the message bus without refactoring their messaging protocols.

  In some cases, it is also possible to integrate an adapter directly into an external application. For example, if you develop an application using FUSE Services Framework, where the service is implemented using JAX-WS and JAXB mappings, it is possible to bind a variety of different transports to the service. These transport bindings function as adapters.

Chapter 4. Message Construction

The message construction patterns describe the various forms and functions of the messages that pass through the system.

Overview

Message construction patterns describe the possible actions that can be performed on messages as they pass through a system. The only formally defined pattern supported by FUSE Mediation Router is the correlation identifier pattern.

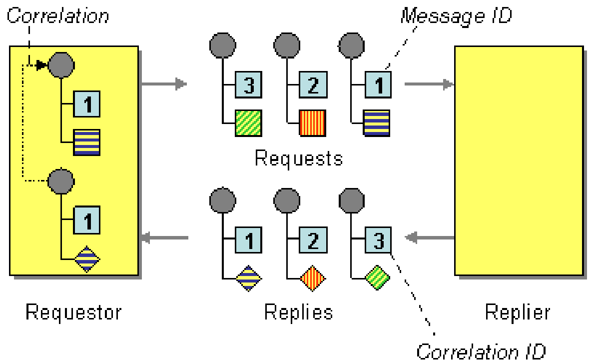
Correlation Identifier

The *correlation identifier* pattern, shown in Figure 4.1 on page 49, describes how to match reply messages with request messages, given that an asynchronous messaging system is used to implement a request-reply protocol. The essence of this idea is that request messages should be generated with a unique token, the *request ID*, that identifies the request message and reply messages should include a token, the *correlation ID*, that contains the matching request ID.

FUSE Mediation Router supports the Correlation Identifier from the EIP patterns by getting or setting a header on a Message.

When working with the ActiveMQ or JMS components, the correlation identifier header is called `JMSCorrelationID`. You can add your own correlation identifier to any message exchange to help correlate messages together in a single conversation (or business process). A correlation identifier is usually stored in a FUSE Mediation Router message header.

Figure 4.1. Correlation Identifier Pattern



Chapter 5. Message Routing

The message routing patterns describe various ways of linking message channels together. This includes various algorithms that can be applied to the message stream (without modifying the body of the message).

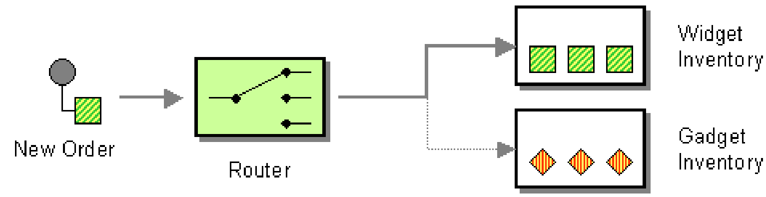
Content-based Router ... 52
Message Filter ... 54
Recipient List ... 56
Splitter ... 59
Aggregator ... 61
Resequencer ... 66
Routing Slip ... 70
Throttler ... 72
Delayer ... 73
Load Balancer ... 75
Multicast ... 79

Content-based Router

Overview

A *content-based router*, shown in Figure 5.1 on page 52, enables you to route messages to the appropriate destination based on the message contents.

Figure 5.1. Content-Based Router Pattern



Java DSL example

The following example shows how to route a request from an input, `seda:a`, endpoint to either `seda:b`, `queue:c`, or `seda:d` depending on the evaluation of various predicate expressions:

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").choice()
          .when(header("foo").isEqualTo("bar")).to("seda:b")
         .when(header("foo").isEqualTo("cheese")).to("seda:c")

          .otherwise().to("seda:d");
    }
};
```

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext id="buildSimpleRouteWithChoice" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <choice>
      <when>
        <xpath>$foo = 'bar'</xpath>
        <to uri="seda:b"/>
      </when>
      <when>
        <xpath>$foo = 'cheese'</xpath>
```

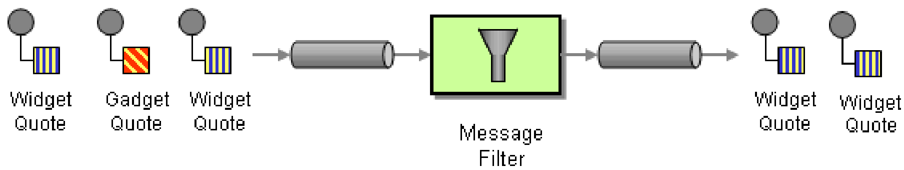
```
        <to uri="seda:c"/>
      </when>
      <otherwise>
        <to uri="seda:d"/>
      </otherwise>
    </choice>
  </route>
</camelContext>
```

Message Filter

Overview

A *message filter* is a processor that eliminates undesired messages based on specific criteria. In FUSE Mediation Router, the message filter pattern, shown in Figure 5.2 on page 54, is implemented by the `filter()` Java DSL command. The `filter()` command takes a single predicate argument, which controls the filter. When the predicate is `true`, the incoming message is allowed to proceed, and when the predicate is `false`, the incoming message is blocked.

Figure 5.2. Message Filter Pattern



Java DSL example

The following example shows how to create a route from endpoint, `seda:a`, to endpoint, `seda:b`, that blocks all messages except for those messages whose `foo` header have the value, `bar`:

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").filter(head
er("foo").isEqualTo("bar")).to("seda:b");
    }
};
```

To evaluate more complex filter predicates, you can invoke one of the supported scripting languages, such as XPath, XQuery, or SQL (see "Languages for Expressions and Predicates" in *Defining Routes*). The following example defines a route that blocks all messages except for those containing a `person` element whose `name` attribute is equal to `James`:

```
from("direct:start").
        filter().xpath("/person[@name='James']").
        to("mock:result");
```

XML configuration example

The following example shows how to configure the route with an XPath predicate in XML (see "Languages for Expressions and Predicates" in *Defining Routes*):

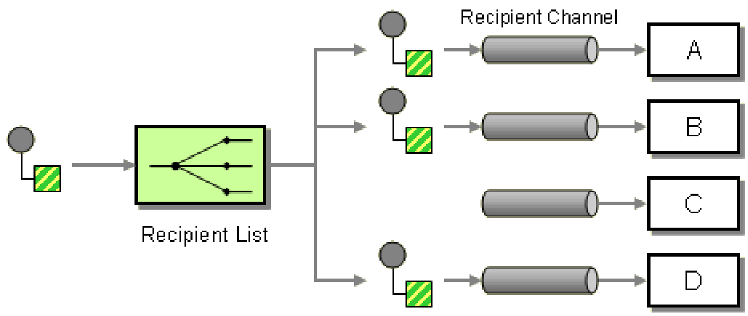
```
<camelContext id="simpleFilterRoute" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <filter>
      <xpath>$foo = 'bar'</xpath>
      <to uri="seda:b"/>
    </filter>
  </route>
</camelContext>
```

Recipient List

Overview

A *recipient list*, shown in Figure 5.3 on page 56, is a type of router that sends each incoming message to multiple different destinations. In addition, a recipient list typically requires that the list of recipients be calculated at run time.

Figure 5.3. Recipient List Pattern



Recipient list with fixed destinations

The simplest kind of recipient list is where the list of destinations is fixed and known in advance, and the exchange pattern is *InOnly*. In this case, you can hardwire the list of destinations into the `to()` Java DSL command.

Note

The examples given here, for the recipient list with fixed destinations, work *only* with the *InOnly* exchange pattern (similar to a pipeline on page 26). If you want to create a recipient list for exchange patterns with *Out* messages, use the multicast pattern instead.

Java DSL example

The following example shows how to route an *InOnly* exchange from a consumer endpoint, `queue:a`, to a fixed list of destinations:

```
from("seda:a").to("seda:b", "seda:c", "seda:d");
```

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext id="buildStaticRecipientList" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:b"/>
    <to uri="seda:c"/>
    <to uri="seda:d"/>
  </route>
</camelContext>
```

Recipient list calculated at run time

In most cases, when you use the recipient list pattern, the list of recipients should be calculated at runtime. To do this use the `recipientList()` processor, which takes a list of destinations as its sole argument. Because FUSE Mediation Router applies a type converter to the list argument, it should be possible to use most standard Java list types (for example, a collection, a list, or an array). For more details about type converters, see "Built-In Type Converters" in *Programmer's Guide*.

Java DSL example

The following example shows how to extract the list of destinations from a message header called `recipientListHeader`, where the header value is a comma-separated list of endpoint URIs:

```
from("direct:a").recipientList(header("recipientListHead
er").tokenize(","));
```

In some cases, if the header value is a list type, you might be able to use it directly as the argument to `recipientList()`. For example:

```
from("seda:a").recipientList(header("recipientListHeader"));
```

However, this example is entirely dependent on how the underlying component parses this particular header. If the component parses the header as a simple string, this example will *not* work. You must know how the underlying component parses its header data (see ????).

XML configuration example

The following example shows how to configure the preceding route in XML, where it is assumed that the underlying component parses the `foo` header as a list type:

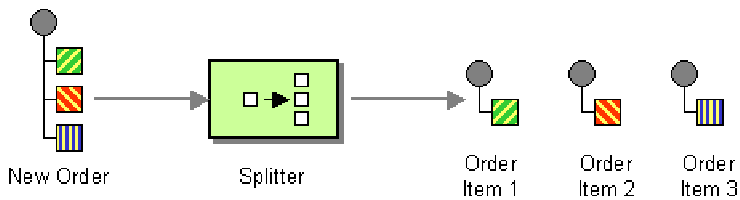
```
<camelContext id="buildDynamicRecipientList" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <recipientList>
      <header>recipientListHeader</header>
    </recipientList>
  </route>
</camelContext>
```

Splitter

Overview

A *splitter* is a type of router that splits an incoming message into a series of outgoing messages. Each of the outgoing messages contains a piece of the original message. In FUSE Mediation Router, the splitter pattern, shown in Figure 5.4 on page 59, is implemented by the `splitter()` Java DSL command, which takes a list of message pieces as its sole argument.

Figure 5.4. Splitter Pattern



Header data

Each outgoing message includes a copy of *all* of the original headers from the incoming message. In addition, the splitter processor adds the headers descibed in Table 5.1 on page 59 to each outgoing message.

Table 5.1. Splitter Headers

Header Name	Type	Description
`org.apache.camel.splitSize`	`Integer`	The total number of parts into which the original message is split.
`org.apache.camel.splitCounter`	`Integer`	Index of the current message part (starting at `0`).

Java DSL example

The following example defines a route from `seda:a` to `seda:b` that splits messages by converting each line of an incoming message into a separate outgoing message:

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").splitter(bodyAs(String.class).token
ize("\n")).to("seda:b");
    }
};
```

The splitter can use any expression language, so you can split messages using any of the supported scripting languages, such as XPath, XQuery, or SQL (see "Languages for Expressions and Predicates" in *Defining Routes*). The following example extracts `bar` elements from an incoming message and insert them into separate outgoing messages:

```
from("activemq:my.queue").split
ter(xpath("//foo/bar")).to("file://some/directory")
```

XML configuration example

The following example shows how to configure a splitter route in XML, using the XPath scripting language:

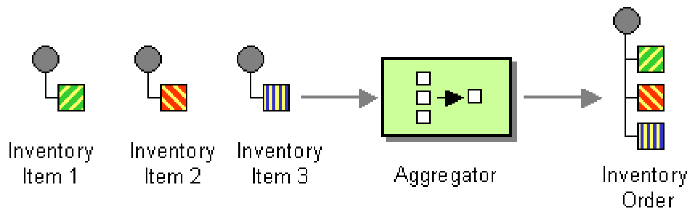
```
<camelContext id="buildSplitter" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
    <route>
      <from uri="seda:a"/>
      <splitter>
        <xpath>//foo/bar</xpath>
        <to uri="seda:b"/>
      </splitter>
    </route>
</camelContext>
```

Aggregator

Overview

The *aggregator* pattern, shown in Figure 5.5 on page 61, enables you to combine a batch of related messages into a single message.

Figure 5.5. Aggregator Pattern



To control the aggregator's behavior, FUSE Mediation Router allows you to specify the properties described in *Enterprise Integration Patterns*, as follows:

- *Correlation expression* — Determines which messages should be aggregated together. The correlation expression is evaluated on each incoming message to produce a *correlation key*. Incoming messages with the same correlation key are then grouped into the same batch. For example, if you want to aggregate *all* incoming messages into a single message, you can use a constant expression.

- *Completeness condition* — Determines when a batch of messages is complete. You can specify this either as a simple size limit or, more generally, you can specify a predicate condition that flags when the batch is complete.

- *Aggregation algorithm* — Combines the message exchanges for a single correlation key into a single message exchange. The default strategy simply chooses the latest message, which makes it ideal for throttling message flows.

For example, consider a stock market data system that receives 30,000 messages per second. You might want to throttle down the message flow if your GUI tool cannot cope with such a massive update rate. The incoming stock quotes can be aggregated together simply by choosing the latest quote and discarding the older prices. (You can apply a delta processing algorithm, if you prefer to capture some of the history.)

Simple aggregator

You can define a simple aggregator by calling the `aggregator()` DSL command with a correlation expression as its sole argument (default limits are applied to the batch size—see "Specifying the batch size" on page 62). The following example shows how to aggregate stock quotes, so that only the latest quote is propagated for the symbol contained in the `StockSymbol` header:

```
from("direct:start").aggregator(header("StockSym
bol")).to("mock:result");
```

The following example shows how to configure the same route using XML configuration:

```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregator>
      <simple>header.StockSymbol</simple>
      <to uri="mock:result"/>
    </aggregator>
  </route>
</camelContext>
```

Specifying the batch size

Normally, you also specify how many messages should be collected (the *batch size*) before the aggregate message gets propagated to the target endpoint. FUSE Mediation Router provides the following settings for controlling the batch size:

- *Batch size* — Specifies an upper limit to the number of messages in a batch (default is 100). For example, the following Java DSL route sets an upper limit of 10 messages in a batch:

  ```
  from("direct:start").aggregator(header("StockSymbol")).batch
  Size(10).to("mock:result");
  ```

  The following example shows how to configure the same route using XML:

  ```
  <camelContext id="camel" xmlns="http://act
  ivemq.apache.org/camel/schema/spring">
    <route>
      <from uri="direct:start"/>
  ```

```
    <aggregator batchSize="10">
      <simple>header.StockSymbol</simple>
      <to uri="mock:result"/>
    </aggregator>
  </route>
</camelContext>
```

- *Batch timeout* — Specifies a time interval, in units of milliseconds, during which messages are collected (default is 1000 ms). If no messages are received during the given time interval, no aggregate message is propagated. For example, the following Java DSL route aggregates the messages that arrive during each ten second window:

  ```
  from("direct:start").aggregator(header("StockSymbol")).batch
  Timeout(10000).to("mock:result");
  ```

  The following example shows how to configure the same route using XML:

  ```
  <camelContext id="camel" xmlns="http://act
  ivemq.apache.org/camel/schema/spring">
    <route>
      <from uri="direct:start"/>
      <aggregator batchTimeout="10000">
        <simple>header.StockSymbol</simple>
        <to uri="mock:result"/>
      </aggregator>
    </route>
  </camelContext>
  ```

- *Completed predicate* — Specifies an arbitrary predicate expression that determines when the current batch is complete. If the predicate resolves to `true`, the current message becomes the last message of the batch. For example, if you want to terminate a batch of stock quotes every time you receive an `ALERT` message (as indicated by the value of a `MsgType` header), you can define a route like the following:

  ```
  from("direct:start").aggregator(header("StockSymbol")).
      completedPredicate(header("Ms
  gType").isEqualTo("ALERT")).to("mock:result");
  ```

  The following example shows how to configure the same route using XML:

  ```
  <camelContext id="camel" xmlns="http://act
  ivemq.apache.org/camel/schema/spring">
    <route>
      <from uri="direct:start"/>
  ```

```
    <aggregator>
      <simple>header.StockSymbol</simple>
      <completedPredicate>
          <xpath>$MsgType = 'ALERT'</xpath>
      </completedPredicate>
      <to uri="mock:result"/>
    </aggregator>
  </route>
</camelContext>
```

You can also combine batch limiting mechanisms, in which case a batch is completed whenever the first of the limits is reached. The following example shows how to specify all three limits simultaneously:

```
from("direct:start").aggregator(header("StockSymbol")).
    batchSize(10).
    batchTimeout(10000).
   completedPredicate(header("MsgType").isEqualTo("ALERT")).

    to("mock:result");
```

Custom aggregation strategy

The default aggregation strategy is to select the most recent message in a batch and discard all the others. If you want to apply a different aggregation strategy, you can implement a custom version of the `org.apache.camel.processor.aggregate.AggregationStrategy` interface, and pass it as the second argument to the `aggregator()` DSL command. For example, to aggregate messages using the custom strategy class, `MyAggregationStrategy`, you can define a route like the following:

```
from("direct:start").aggregator(header("StockSymbol"), new
MyAggregationStrategy()).to("mock:result");
```

The following code implements a custom aggregation strategy, `MyAggregationStrategy`, that concatenates all of the batch messages into a single, large message:

```
// Java
package com.my_package_name

import org.apache.camel.processor.aggregate.Aggregation
Strategy;
import org.apache.camel.Exchange;

public class MyAggregationStrategy implements Aggregation
Strategy {
    public Exchange aggregate(Exchange oldExchange, Exchange
```

```
 newExchange) {
        String oldBody = oldExchange.getIn().get
Body(String.class);
        String newBody = newExchange.getIn().get
Body(String.class);
        String concatBody = oldBody.concat(newBody);
        // Set the body equal to a concatenation of old and
new.
        oldExchange.getIn().setBody(concatBody);
        // Ignore the message headers!
        // (in a real application, you would probably want to
 do
       //  something more sophisticated with the header data).

        return oldExchange;
    }
}
```

You can also configure a route with a custom aggregation strategy in XML, as follows:

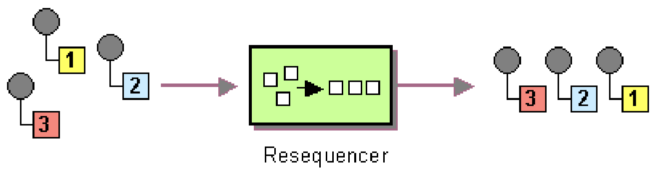
```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregator strategyRef="aggregatorStrategy">
      <simple>header.StockSymbol</simple>
      <to uri="mock:result"/>
    </aggregator>
  </route>
</camelContext>

<bean id="aggregatorStrategy" class="com.my_package_name.MyAg
gregationStrategy"/>
```

Resequencer

Overview

The *resequencer* pattern, shown in Figure 5.6 on page 66, enables you to resequence messages according to a sequencing expression. Messages that generate a low value for the sequencing expression are moved to the front of the batch and messages that generate a high value are moved to the back.

Figure 5.6. Resequencer Pattern



FUSE Mediation Router supports two resequencing algorithms:

- *Batch resequencing* — Collects messages into a batch, sorts the messages and sends them to their output.
- *Stream resequencing* — Re-orders (continuous) message streams based on the detection of gaps between messages.

Batch resequencing

The batch resequencing algorithm is enabled by default. For example, to resequence a batch of incoming messages based on the value of a timestamp contained in the `TimeStamp` header, you can define the following route in Java DSL:

```
from("direct:start").resequencer(head
er("TimeStamp")).to("mock:result");
```

By default, the batch is obtained by collecting all of the incoming messages that arrive in a time interval of 1000 milliseconds (default *batch timeout*), up to a maximum of 100 messages (default *batch size*). You can customize the values of the batch timeout and the batch size by appending a `batch()` DSL command, which takes a `BatchResequencerConfig` instance as its sole argument. For example, to modify the preceding route so that the batch consists of messages collected in a 4000 millisecond time window, up to a maximum of 300 messages, you can define the Java DSL route as follows:

```
import org.apache.camel.model.config.BatchResequencerConfig;

RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start").resequencer(head
er("TimeStamp")).batch(new BatchResequencerCon
fig(300,4000L)).to("mock:result");
    }
};
```

You can also use multiple expressions to sort messages in a batch. For example, if you want to sort incoming messages, first, according to their JMS priority (as recorded in the `JMSPriority` header) and second, according to the value of their time stamp (as recorded in the `TimeStamp` header), you can define a route like the following:

```
from("direct:start").resequencer(header("JMSPriority"), head
er("TimeStamp")).to("mock:result");
```

In this case, messages with the highest priority (that is, low JMS priority number) are moved to the front of the batch. If more than one message has the highest priority, the highest priority messages are put in order according to the value of the `TimeStamp` header.

You can also specify a batch resequencer pattern using XML configuration. The following example defines a batch resequencer with a batch size of 300 and a batch timeout of 4000 milliseconds:

```
<camelContext id="resequencerBatch" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start" />
    <resequencer>
      <simple>header.TimeStamp</simple>
      <to uri="mock:result" />
      <!--
        batch-config can be omitted for default (batch)
resequencer settings
      -->
      <batch-config batchSize="300" batchTimeout="4000" />
    </resequencer>
  </route>
</camelContext>
```

Stream resequencing

To enable the stream resequencing algorithm, you must append `stream()` to the `resequencer()` DSL command. For example, to resequence incoming messages based on the value of a sequence number in the `seqnum` header, you define a DSL route as follows:

```
from("direct:start").resequencer(header("se
qnum")).stream().to("mock:result");
```

The stream-processing resequencer algorithm is based on the detection of gaps in a message stream, rather than on a fixed batch size. Gap detection, in combination with timeouts, removes the constraint of needing to know the number of messages of a sequence (that is, the batch size) in advance. Messages must contain a unique sequence number for which a predecessor and a successor is known. For example a message with the sequence number `3` has a predecessor message with the sequence number `2` and a successor message with the sequence number `4`. The message sequence `2,3,5` has a gap because the successor of `3` is missing. The resequencer therefore must retain message `5` until message `4` arrives (or a timeout occurs).

By default, the stream resequencer is configured with a timeout of 1000 milliseconds, and a maximum message capacity of 100. To customize the stream's timeout and message capacity, you can pass a `StreamResequencerConfig` object as an argument to `stream()`. For example, to configure a stream resequencer with a message capacity of 5000 and a timeout of 4000 milliseconds, you define a route as follows:

```
// Java
import org.apache.camel.model.config.StreamResequencerConfig;

RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start").resequencer(header("seqnum")).
            stream(new StreamResequencerConfig(5000, 4000L)).

            to("mock:result");
    }
};
```

If the maximum time delay between successive messages (that is, messages with adjacent sequence numbers) in a message stream is known, the resequencer's timeout parameter should be set to this value. In this case, you can guarantee that all messages in the stream are delivered in the correct order to the next processor. The lower the timeout value that is compared to the out-of-sequence time difference, the more likely it is that the resequencer will deliver messages out of sequence. Large timeout values should be supported by sufficiently high capacity values, where the capacity parameter is used to prevent the resequencer from running out of memory.

If you want to use sequence numbers of some type other than `long`, you would must define a custom comparator, as follows:

```
// Java
ExpressionResultComparator<Exchange> comparator = new MyCompar
ator();
StreamResequencerConfig config = new StreamResequencerCon
fig(5000, 4000L, comparator);
from("direct:start").resequencer(header("seqnum")).stream(con
fig).to("mock:result");
```

You can also specify a stream resequencer pattern using XML configuration. The following example defines a stream resequencer with a message capacity of 5000 and a timeout of 4000 milliseconds:

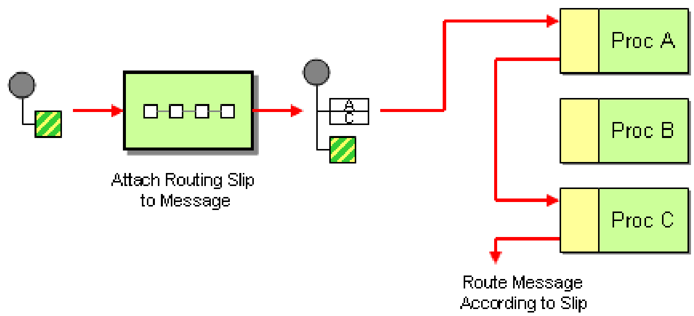
```
<camelContext id="resequencerStream" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <resequencer>
      <simple>header.seqnum</simple>
      <to uri="mock:result" />
      <stream-config capacity="5000" timeout="4000"/>
    </resequencer>
  </route>
</camelContext>
```

Routing Slip

Overview

The *routing slip* pattern, shown in Figure 5.7 on page 70, enables you to route a message consecutively through a series of processing steps, where the sequence of steps is not known at design time and can vary for each message. The list of endpoints through which the message should pass is stored in a header field (the *slip*), which FUSE Mediation Router reads at run time to construct a pipeline on the fly.

Figure 5.7. Routing Slip Pattern



The slip header

By default, the routing slip appears in a header named `routingSlipHeader`, where the header value is a comma-separated list of endpoint URIs. For example, a routing slip that specifies a sequence of security tasks—decrypting, authenticating, and de-duplicating a message—might look like the following:

```
cxf:bean:decrypt,cxf:bean:authenticate,cxf:bean:dedup
```

Java DSL example

The following route takes messages from the `direct:a` endpoint and passes them into the routing slip pattern:

```
from("direct:a").routingSlip();
```

You can customize the name of the routing slip header by passing a string argument to the `routingSlip()` command, as follows:

```
from("direct:b").routingSlip("aRoutingSlipHeader");
```

You can also customize the URI delimiter using the two-argument form of `routingSlip()`. The following example defines a route that customizes the routing slip header to be `aRoutingSlipHeader` and, to specify `#` as the URI delimiter:

```
from("direct:c").routingSlip("aRoutingSlipHeader", "#");
```

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext id="buildRoutingSlip" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:c"/>
    <routingSlip headerName="aRoutingSlipHeader" uriDelim
iter="#"/>
  </route>
</camelContext>
```

Throttler

Overview

A *throttler* is a processor that limits the flow rate of incoming messages. You can use this pattern to protect a target endpoint from getting overloaded. In FUSE Mediation Router, you can implement the throttler pattern using the `throttler()` Java DSL command.

Java DSL example

To limit the flow rate to 100 messages per second, define a route as follows:

```
from("seda:a").throttler(100).to("seda:b");
```

If necessary, you can customize the time period that governs the flow rate using the `timePeriodMillis()` DSL command. For example, to limit the flow rate to 3 messages per 30000 milliseconds, define a route as follows:

```
from("seda:a").throttler(3).timePeriodMil
lis(30000).to("mock:result");
```

XML configuration example

The following example shows how to configure the preceding route in XML:

```
<camelContext id="throttlerRoute" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <throttler maximumRequestsPerPeriod="3" timePeriodMil
lis="30000">
       <to uri="mock:result"/>
    </throttler>
  </route>
</camelContext>
```

Delayer

Overview

A *delayer* is a processor that enables you to apply either a *relative* time delay or an *absolute* time delay to incoming messages.

Java DSL example

You can use the `delayer()` command to add a *relative* time delay, in units of milliseconds, to incoming messages. For example, the following route delays all incoming messages by 2 seconds:

```
from("seda:a").delayer(2000).to("mock:result");
```

Alternatively, you can specify the *absolute* time when a message should be dispatched. The absolute time value must be expressed in coordinated universal time (UTC), which is defined as the number of milliseconds that have elapsed since midnight, January 1, 1970. For example, to dispatch a message at the absolute time specified by the contents of the `JMSTimestamp` header, you can define a route like the following:

```
from("seda:a").delayer(header("JMSTimestamp")).to("mock:res
ult");
```

You can also combine an absolute time with a relative time delay. For example, to delay an incoming message until the time specified in the `JMSTimestamp` header plus an additional 3 seconds, you define a route as follows:

```
from("seda:a").delayer(header("JMSTimestamp"),
3000).to("mock:result");
```

The preceding examples assume that delivery order is maintained. However, this could result in messages being delivered later than their specified time stamp. To avoid this, you can reorder the messages based on their delivery time, by combining the delayer pattern with the resequencer pattern. For example:

```
from("activemq:someQueue").
    resequencer(header("JMSTimestamp")).
    delayer(header("JMSTimestamp")).to("activemq:aD
elayedQueue");
```

XML configuration example

To delay an incoming message until the time specified in the `JMSTimestamp` header plus an additional 3 seconds, define a route using the following XML configuration:

```
<camelContext id="delayerRoute" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <delayer>
      <simple>header.JMSTimestamp</simple>
      <delay>3000</delay>
    </delayer>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

If you want to specify a relative time delay only, you must insert a dummy expression, `<expression/>`, in place of the absolute time expression. For example, to delay incoming messages by a relative time delay of 2 seconds, define a route as follows:

```
<camelContext id="delayerRoute2" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <delayer>
      <expression/>
      <delay>2000</delay>
    </delayer>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

Load Balancer

Overview

The *load balancer* pattern allows you to delegate message processing to one of several endpoints, using a variety of different load-balancing policies.

Java DSL example

The following route distributes incoming messages between the target endpoints, `mock:x`, `mock:y`, `mock:z`, using a round robin load-balancing policy:

```
from("direct:start").loadBalance().roundRobin().to("mock:x",
 "mock:y", "mock:z");
```

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
        <roundRobin/>
        <to uri="mock:x"/>
        <to uri="mock:y"/>
        <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

Note

In versions of FUSE Mediation Router earlier than 1.4.2.0, the `<roundRobin/>` tag must appear as the *last* tag inside the `loadBalance` element.

Load-balancing policies

The FUSE Mediation Router load balancer supports the following load-balancing policies:

- "Round robin"
- "Random"
- "Sticky"

- "Topic"

Round robin

The round robin load-balancing policy cycles through all of the target endpoints, sending each incoming message to the next endpoint in the cycle. For example, if the list of target endpoints is, `mock:x`, `mock:y`, `mock:z`, then the incoming messages are sent to the following sequence of endpoints: `mock:x`, `mock:y`, `mock:z`, `mock:x`, `mock:y`, `mock:z`, and so on.

You can specify the round robin load-balancing policy in Java DSL, as follows:

```
from("direct:start").loadBalance().roundRobin().to("mock:x",
 "mock:y", "mock:z");
```

Alternatively, you can configure the same route in XML, as follows:

```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
        <roundRobin/>
        <to uri="mock:x"/>
        <to uri="mock:y"/>
        <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

Random

The random load-balancing policy chooses the target endpoint randomly from the specified list.

You can specify the random load-balancing policy in Java DSL, as follows:

```
from("direct:start").loadBalance().random().to("mock:x",
"mock:y", "mock:z");
```

Alternatively, you can configure the same route in XML, as follows:

```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
        <random/>
        <to uri="mock:x"/>
```

```
        <to uri="mock:y"/>
        <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

Sticky

The sticky load-balancing policy directs the *In* message to an endpoint that is chosen by calculating a hash value from a specified expression. The advantage of this load-balancing policy is that expressions of the same value are always sent to the same server. For example, by calculating the hash value from a header that contains a username, you ensure that messages from a particular user are always sent to the same target endpoint. Another useful approach is to specify an expression that extracts the session ID from an incoming message. This ensures that all messages belonging to the same session are sent to the same target endpoint.

You can specify the sticky load-balancing policy in Java DSL, as follows:

```
from("direct:start").loadBalance().sticky(header("user
name")).to("mock:x", "mock:y", "mock:z");
```

Alternatively, you can configure the same route in XML, as follows:

```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
      <sticky>
        <expression>
          <simple>header.username</simple>
        </expression>
      </sticky>
      <to uri="mock:x"/>
      <to uri="mock:y"/>
      <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

Topic

The topic load-balancing policy sends a copy of each *In* message to *all* of the listed destination endpoints (effectively broadcasting the message to all of the destinations, like a JMS topic).

You can use the Java DSL to specify the topic load-balancing policy, as follows:

```
from("direct:start").loadBalance().topic().to("mock:x",
"mock:y", "mock:z");
```

Alternatively, you can configure the same route in XML, as follows:

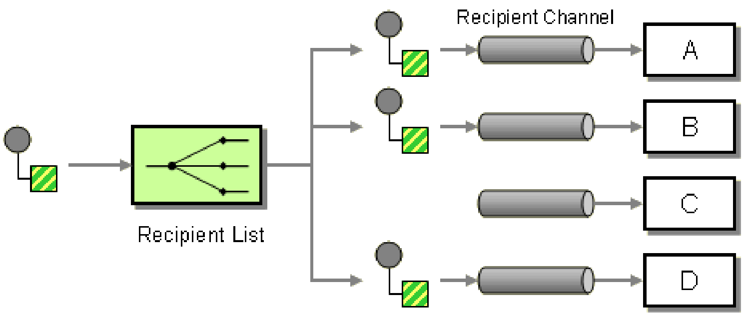
```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
        <topic/>
        <to uri="mock:x"/>
        <to uri="mock:y"/>
        <to uri="mock:z"/>
    </loadBalance>
  </route>
</camelContext>
```

Multicast

Overview

The *multicast* pattern, shown in Figure 5.8 on page 79, is a variation of the recipient list with a fixed destination pattern, which is compatible with the *InOut* message exchange pattern. This is in contrast to recipient list, which is only compatible with the *InOnly* exchange pattern.

Figure 5.8. Multicast Pattern



Multicast with a custom aggregation strategy

Whereas the multicast processor receives multiple *Out* messages in response to the original request (one from each of the recipients), the original caller is only expecting to receive a *single* reply. Thus, there is an inherent mismatch on the reply leg of the message exchange, and to overcome this mismatch, you must provide a custom *aggregation strategy* to the multicast processor. The aggregation strategy class is responsible for aggregating all of the *Out* messages into a single reply message.

Consider the example of an electronic auction service, where a seller offers an item for sale to a list of buyers. The buyers each put in a bid for the item, and the seller automatically selects the bid with the highest price. You can implement the logic for distributing an offer to a fixed list of buyers using the `multicast()` DSL command, as follows:

```
from("cxf:bean:offer").multicast(new HighestBidAggregation
Strategy()).
    to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buy
er3");
```

Where the seller is represented by the endpoint, `cxf:bean:offer`, and the buyers are represented by the endpoints, `cxf:bean:Buyer1`, `cxf:bean:Buyer2`, `cxf:bean:Buyer3`. To consolidate the bids received from the various buyers, the multicast processor uses the aggregation strategy, `HighestBidAggregationStrategy`. You can implement the `HighestBidAggregationStrategy` in Java, as follows:

```
// Java
import org.apache.camel.processor.aggregate.Aggregation
Strategy;
import org.apache.camel.Exchange;

public class HighestBidAggregationStrategy implements Aggreg
ationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange
 newExchange) {
        float oldBid = oldExchange.getOut().getHeader("Bid",
 Float.class);
        float newBid = newExchange.getOut().getHeader("Bid",
 Float.class);
        return (newBid > oldBid) ? newExchange : oldExchange;

    }
}
```

Where it is assumed that the buyers insert the bid price into a header named, `Bid`. For more details about custom aggregation strategies, see "Aggregator" on page 61.

Parallel processing

By default, the multicast processor invokes each of the recipient endpoints one after another (in the order listed in the `to()` command). In some cases, this might cause unacceptably long latency. To avoid these long latency times, you have the option of enabling parallel processing in the multicast processor by passing the value `true` as the second argument. For example, to enable parallel processing in the electronic auction example, define the route as follows:

```
from("cxf:bean:offer")
    .multicast(new HighestBidAggregationStrategy(), true)
    .to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buy
er3");
```

Where the multicast processor now invokes the buyer endpoints, using a thread pool that has one thread for each of the endpoints.

If you want to customize the size of the thread pool that invokes the buyer endpoints, you can invoke the `setThreadPoolExecutor()` method to specify your own custom thread pool executor. For example:

```
from("cxf:bean:offer")
    .multicast(new HighestBidAggregationStrategy(), true)
    .setThreadPoolExecutor(MyExecutor)
    .to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buy
er3");
```

Where *MyExecutor* is an instance of java.util.concurrent.ThreadPoolExecutor[1] type.

XML configuration example

The following example shows how to configure a similar route in XML, where the route uses a custom aggregation strategy and a custom thread executor:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans ht
tp://www.springframework.org/schema/beans/spring-beans-2.5.xsd

       http://activemq.apache.org/camel/schema/spring ht
tp://activemq.apache.org/camel/schema/spring/camel-spring.xsd

    ">

  <camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
    <route>
      <from uri="cxf:bean:offer"/>
      <multicast strategyRef="highestBidAggregationStrategy"
                 parallelProcessing="true"
                 threadPoolRef="myThreadExcutor">
         <to uri="cxf:bean:Buyer1"/>
         <to uri="cxf:bean:Buyer2"/>
         <to uri="cxf:bean:Buyer3"/>
      </multicast>
    </route>
  </camelContext>

  <bean id="highestBidAggregationStrategy"
class="com.acme.example.HighestBidAggregationStrategy"/>
  <bean id="myThreadExcutor" class="com.acme.example.MyThreadEx
cutor"/>

</beans>
```

[1] http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ThreadPoolExecutor.html

Where both the `parallelProcessing` attribute and the `threadPoolRef` attribute are optional. It is only necessary to set them if you want to customize the threading behavior of the multicast processor.

Chapter 6. Message Transformation

The message transformation patterns describe how to modify the contents of messages for various purposes.

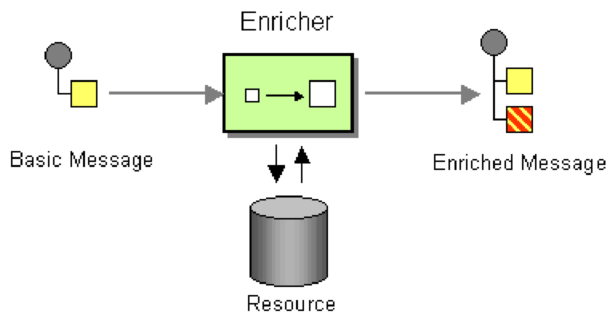
Content Enricher 84
Content Filter 86
Normalizer 88

Content Enricher

Overview

The *content enricher* pattern describes a scenario where the message destination requires more data than is present in the original message. In this case, you would use a content enricher to pull in the extra data from an external resource.

Figure 6.1. Content Enricher Pattern



Implementing a content enricher

You can use one of the following approaches to implement a content enricher:

- *Templating* — A scripting technique that involves extracting portions of a message and inserting them into a given template. FUSE Mediation Router supports templating with several different scripting languages and components. See Templating[1] for details.
- *Bean integration* — Enables you to call any method on a registered bean. The bean method can modify the message to enrich the content. See Bean integration[2] for details.

Java DSL example

You can use templating to consume a message from one destination, transform it with a scripting language, like Velocity or XQuery, and then send it on to another destination. For example, using the *InOnly* exchange pattern (one way messaging):

[1] http://activemq.apache.org/camel/templating.html
[2] http://activemq.apache.org/camel/bean-integration.html

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

If you want to use *InOut* (request-reply) semantics to process requests on the `My.Queue` queue in ActiveMQ with a template-generated response and then send responses back to the `JMSReplyTo` destination, you could define a route like the following:

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

For more details about the Velocity component, see ????.

XML configuration example

The following example shows how to configure the same route in XML:

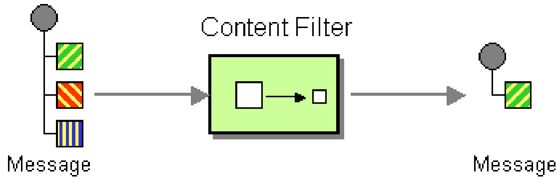
```
<camelContext xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="activemq:My.Queue"/>
    <to uri="velocity:com/acme/MyResponse.vm"/>
    <to uri="activemq:Another.Queue"/>
  </route>
</camelContext>
```

Content Filter

Overview

The *content filter* pattern describes a scenario where you need to filter out extraneous content from a message before delivering it to its intended recipient. For example, you might employ a content filter to strip out confidential information from a message.

Figure 6.2. Content Filter Pattern



A common way to filter messages is to use an expression in the DSL, written in one of the supported scripting languages (for example, XSLT, XQuery or JoSQL).

Implementing a content filter

A content filter is essentially an application of a message processing technique for a particular purpose. To implement a content filter, you can employ any of the following message processing techniques:

- *Message translator*—see message translators on page 31.
- *Processors*—see "Implementing a Processor" in *Programmer's Guide*.
- Bean integration[3].

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="activemq:My.Queue"/>
    <to uri="xslt:classpath:com/acme/content_filter.xsl"/>
    <to uri="activemq:Another.Queue"/>
```

[3] http://activemq.apache.org/camel/bean-integration.html

```
  </route>
</camelContext>
```

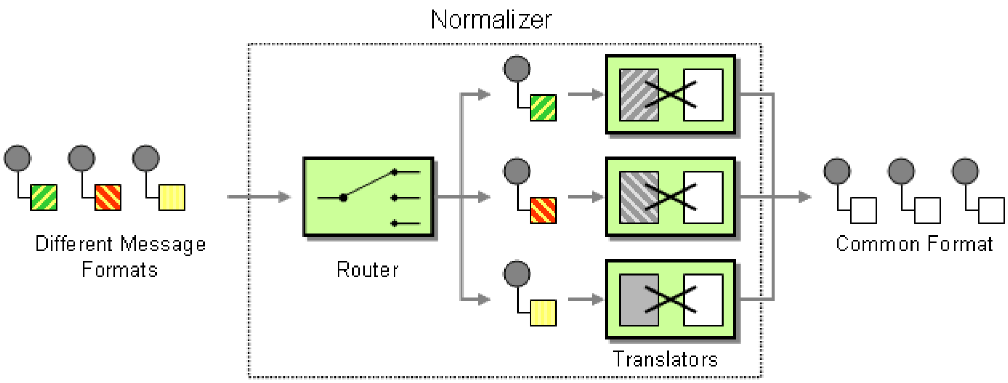
Normalizer

Overview

The *normalizer* pattern is used to process messages that are semantically equivalent, but arrive in different formats. The normalizer transforms the incoming messages into a common format.

In FUSE Mediation Router, you can implement the normalizer pattern by combining a content-based router on page 52, which detects the incoming message's format, with a collection of different message translators on page 31, which transform the different incoming formats into a common format.

Figure 6.3. Normalizer Pattern



Chapter 7. Messaging Endpoints

Messaging Mapper 90
Event Driven Consumer 92
Polling Consumer 93
Competing Consumers 95
Message Dispatcher 98
Selective Consumer 101
Durable Subscriber 104
Idempotent Consumer 107
Transactional Client 110
Messaging Gateway 114
Service Activator 115

Messaging Mapper

Overview

The *messaging mapper* pattern describes how to map domain objects cleanly to and from a canonical message format.

The purpose of the messaging mapper pattern is to create a clean mapping from domain objects to a canonical message format, where the message format is chosen to be as platform neutral as possible. In other words, the chosen message format should be suitable for transmission through a message bus on page 47, where the message bus is the backbone for integrating a variety of different systems, some of which might not be object-oriented.

Many different approaches are possible, but not all of them are clean enough to fulfill the requirements of a messaging mapper. For example, an obvious way to transmit an object would be to use *object serialization*, which enables you to write an object to a data stream using an unambiguous encoding (supported natively in Java). This would *not* be a suitable approach to use for the messaging mapper pattern, however, because the serialization format is understood only by Java applications. Java object serialization would create an impedance mismatch between the original application and the other applications in the messaging system.

The requirements on a messaging mapper can be summarized as follows:

- The canonical message format used to transmit domain objects should be suitable for consumption by non-object oriented applications.
- The mapper code should be implemented separately from the domain object code and separately from the messaging infrastructure. FUSE Mediation Router helps you to fulfill this requirement by providing hooks that can be used to insert mapper code into a route.
- The mapper might need to find an effective way of dealing with certain object-oriented concepts such as inheritance, object references, and object trees. The complexity of these issues will vary from application to application, but the aim of the mapper implementation must always be to create messages that can be processed effectively by non-object-oriented applications.

Finding objects to map

You could use one of the following mechanisms to find the objects to map:

- *Find a registered bean* — For singleton objects and small numbers of objects, you could use the `CamelContext` registry to store references to beans. For example, if a bean instance is instantiated using Spring XML, it is automatically entered into the registry, where the bean is identified by the value of its `id` attribute.

- *Select objects using the JoSQL language* — If all of the objects you want to access are already instantiated at runtime, you could use the JoSQL language to locate a specific object (or objects). For example, if you have a class, `org.apache.camel.builder.sql.Person`, with a `name` bean property and the incoming message has a `UserName` header, you could select the object whose `name` property equals the value of the `UserName` header using the following code:

```
// Java
import static org.apache.camel.builder.sql.SqlBuilder.sql;
import org.apache.camel.Expression;
...
Expression expression = sql("SELECT * FROM
org.apache.camel.builder.sql.Person where name = :UserName");
Object value = expression.evaluate(exchange);
```

  Where the syntax, `:HeaderName`, is used to substitute the value of a header in a JoSQL expression.

- *Dynamic* — For a more scalable solution, it might be necessary to read object data from a database. In some cases, the existing object-oriented application might already provide a finder object that can load objects from the database. In other cases, you might have to write some custom code to extract objects from a database: the JDBC component and the SQL component might be useful in these cases.

Event Driven Consumer

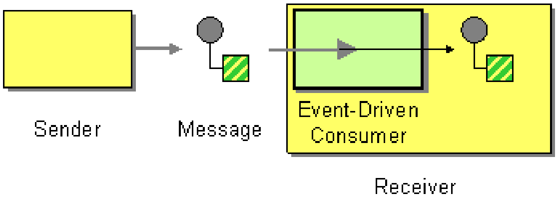
Overview

The *event-driven consumer* pattern is a pattern for implementing the consumer endpoint in a FUSE Mediation Router component and is thus only relevant to programmers who need to develop a custom component in FUSE Mediation Router. Existing components already have a consumer implementation pattern hard-wired into them.

Consumers that conform to this pattern provide an event method that is automatically called by the messaging channel or transport layer whenever an incoming message is received. One of the characteristics of the event-driven consumer pattern is that the consumer endpoint itself does not provide any threads to process the incoming messages. Instead, the underlying transport or messaging channel implicitly provides a processor thread when it invokes the exposed event method (which blocks for the duration of the message processing).

For more details about this implementation pattern, see "Consumer Patterns and Threading" in *Programmer's Guide* and "Consumer Interface" in *Programmer's Guide*.

Figure 7.1. Event Driven Consumer Pattern



Polling Consumer

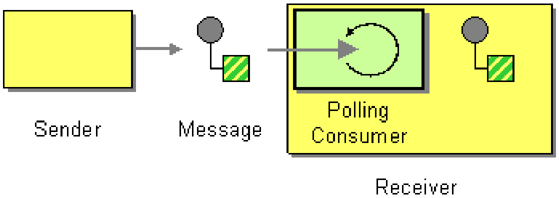
Overview

The *polling consumer* pattern is a pattern for implementing the consumer endpoint in a FUSE Mediation Router component and is thus only relevant to programmers who need to develop a custom component in FUSE Mediation Router. Existing components already have a consumer implementation pattern hard-wired into them.

Consumers that conform to this pattern expose polling methods, `receive()`, `receive(long timeout)`, and `receiveNoWait()` that return a new exchange object, if one is available from the monitored resource. A polling consumer implementation must provide its own thread pool to perform the polling.

For more details about this implementation pattern, see "Consumer Patterns and Threading" in *Programmer's Guide* and "Consumer Interface" in *Programmer's Guide*.

Figure 7.2. Polling Consumer Pattern



Scheduled poll consumer

Many of the FUSE Mediation Router consumer endpoints employ a scheduled poll pattern to receive messages at the start of a route. That is, the endpoint appears to implement an event-driven consumer interface, but internally a scheduled poll is used to monitor a resource that provides the incoming messages for the endpoint.

See "Implementing the Consumer Interface" in *Programmer's Guide* for details of how to implement this pattern.

Quartz component

You can use the quartz component to provide scheduled delivery of messages using the *Quartz* enterprise scheduler. See ???? and Quartz Component[1] for details.

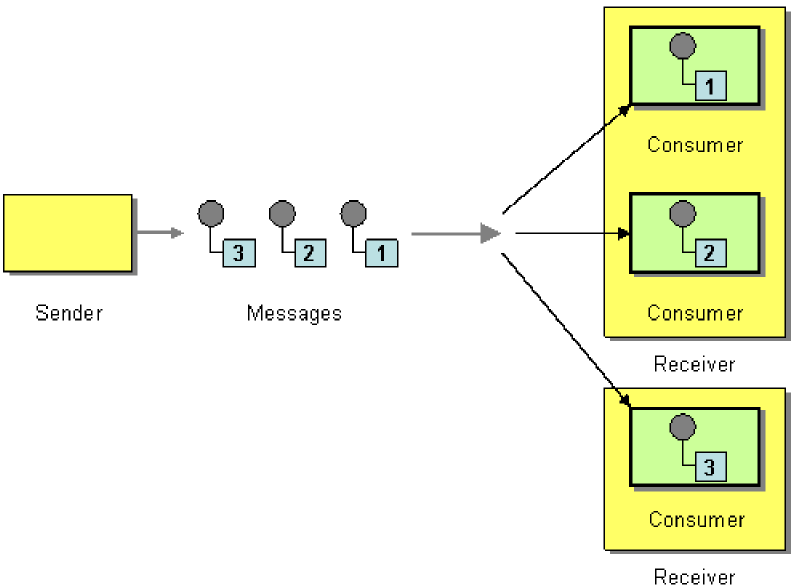
[1] http://activemq.apache.org/camel/quartz.html

Competing Consumers

Overview

The *competing consumers* pattern enables multiple consumers to pull messages off the same queue, with the guarantee that *each message is consumed once only*. This pattern can therefore be used to replace serial message processing with concurrent message processing (bringing a corresponding reduction in response latency).

Figure 7.3. Competing Consumers Pattern



The following components demonstrate the competing consumers pattern:

- "JMS based competing consumers" on page 95
- "SEDA based competing consumers" on page 96

JMS based competing consumers

A regular JMS queue implicitly guarantees that each message can be consumed at most once. Hence, a JMS queue automatically supports the competing consumers pattern. For example, you could define three competing consumers that pull messages from the JMS queue, `HighVolumeQ`, as follows:

```
from("jms:HighVolumeQ").to("cxf:bean:replica01");
from("jms:HighVolumeQ").to("cxf:bean:replica02");
from("jms:HighVolumeQ").to("cxf:bean:replica03");
```

Where the CXF (Web services) endpoints, `replica01`, `replica02`, and `replica03`, process messages from the `HighVolumeQ` queue in parallel.

Alternatively, you can set the JMS query option, `concurrentConsumers`, in order to create a thread pool of competing consumers. For example, the following route creates a pool of three competing threads that pick messages off the specified queue:

```
from("jms:HighVolumeQ?concurrentConsumers=3").to("cxf:bean:rep
lica01");
```

Note

JMS topics *cannot* support the competing consumers pattern. By definition, a JMS topic is intended to send multiple copies of the same message to different consumers. It is, therefore, incompatible with the competing consumers pattern.

SEDA based competing consumers

The purpose of the SEDA component to simplify concurrent processing by breaking the computation up into stages. A SEDA endpoint essentially encapsulates an in-memory blocking queue (implemented by `java.util.concurrent.BlockingQueue`). You can, therefore, use a SEDA endpoint to break a route up into stages, where each stage might use multiple threads. For example, you can define a SEDA route consisting of two stages, as follows:

```
// Stage 1: Read messages from file system.
from("file://var/messages").to("seda:fanout");

// Stage 2: Perform concurrent processing (3 threads).
from("seda:fanout").to("cxf:bean:replica01");
from("seda:fanout").to("cxf:bean:replica02");
from("seda:fanout").to("cxf:bean:replica03");
```

Where the first stage contains a single thread that consumes message from a file endpoint, `file://var/messages`, and routes them to a SEDA endpoint, `seda:fanout`. The second stage contains three threads: a thread that routes exchanges to `cxf:bean:replica01`, a thread that routes exchanges to `cxf:bean:replica02`, and a thread that routes exchanges to `cxf:bean:replica03`. These three threads compete to take exchange instances from the SEDA endpoint, which is implemented using a blocking queue. Because the blocking queue uses locking to prevent more than one thread accessing the queue at a time, you are guaranteed that each exchange instance is consumed at most once.

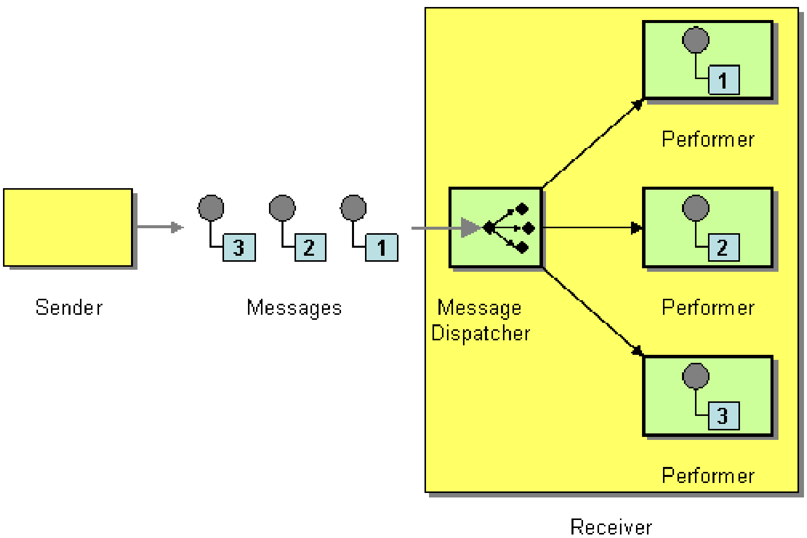
For a discussion of the differences between a SEDA endpoint and a thread pool created by `thread()`, see ????.

Message Dispatcher

Overview

The *message dispatcher* pattern is used to consume messages from a channel and distribute them locally to *performers*, which are responsible for processing the messages. In a FUSE Mediation Router application, performers are usually represented by in-process endpoints, which are used to transfer messages to another section of the route.

Figure 7.4. Message Dispatcher Pattern



You can implement the message dispatcher pattern in FUSE Mediation Router using one of the following approaches:

- "JMS selectors" on page 98.
- "JMS selectors in ActiveMQ" on page 100.
- "Content-based router" on page 100.

JMS selectors

If your application consumes messages from a JMS queue, you can implement the message dispatcher pattern using *JMS selectors*. A JMS selector is a predicate expression involving JMS headers and JMS properties: if the selector evaluates to `true`, the JMS message is allowed to reach the consumer; if the selector evaluates to `false`, the JMS message is blocked. In many respects, a JMS selector is like a filter processor on page 54, but it has the added advantage that the filtering is implemented inside the JMS provider. This means that a JMS selector can block messages before they are transmitted to the FUSE Mediation Router application, giving a significant efficiency advantage.

In FUSE Mediation Router, you can define a JMS selector on a consumer endpoint by setting the `selector` query option on a JMS endpoint URI. For example:

```
from("jms:dispatcher?selector=Country
Code='US'").to("cxf:bean:replica01");
from("jms:dispatcher?selector=Country
Code='IE'").to("cxf:bean:replica02");
from("jms:dispatcher?selector=Country
Code='DE'").to("cxf:bean:replica03");
```

Where the predicates that appear in a selector string are based on a subset of the SQL92 conditional expression syntax (for full details, see the JMS specification[2]). The identifiers appearing in a selector string can refer either to JMS headers or to JMS properties. For example, in the preceding routes, we presume that the sender has set a JMS property called `CountryCode`.

If you want to add a JMS property to a message from within your FUSE Mediation Router application, you can do so by setting a message header (either on *In* message or on *Out* messages). When reading or writing to JMS endpoints, FUSE Mediation Router maps JMS headers and JMS properties to and from its native message headers.

Technically, the selector strings must be URL encoded according to the `application/x-www-form-urlencoded` MIME format (see the HTML specification[3]). In practice, however, the only character that might cause difficulties is `&` (ampersand), because this character is used to delimit each query option in the URI. For more complex selector strings that might need to embed the `&` character, you can encode the strings using the `java.net.URLEncoder` utility class. For example:

```
from("jms:dispatcher?selector=" + java.net.URLEncoder.en
code("CountryCode='US'","UTF-8")).
    to("cxf:bean:replica01");
```

[2] http://java.sun.com/products/jms/docs.html
[3] http://www.w3.org/TR/html4/

Where the UTF-8 encoding must be used.

JMS selectors in ActiveMQ

You can also define JMS selectors on ActiveMQ endpoints. For example:

```
from("activemq:dispatcher?selector=Country
Code='US'").to("cxf:bean:replica01");
from("activemq:dispatcher?selector=Country
Code='IE'").to("cxf:bean:replica02");
from("activemq:dispatcher?selector=Country
Code='DE'").to("cxf:bean:replica03");
```

For more details, see ActiveMQ: JMS Selectors[4] and ActiveMQ Message Properties[5].

Content-based router

The essential difference between the content-based router pattern and the message dispatcher pattern is that a content-based router dispatches messages to physically separate destinations (remote endpoints), whereas a message dispatcher dispatches messages locally, within the same process space. In FUSE Mediation Router, the distinction between these two patterns is not very great, because the same router logic can be used to implement both of them. The only distinction is whether the target endpoints are remote (content-based router) or in-process (message dispatcher).

For details and examples of how to use the content-based router pattern see "Content-based Router" on page 52.

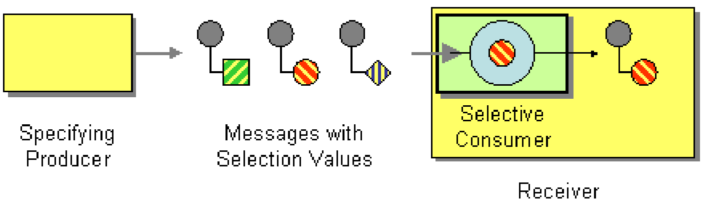
[4] http://activemq.apache.org/selectors.html
[5] http://activemq.apache.org/activemq-message-properties.html

Selective Consumer

Overview

The *selective consumer* pattern describes a consumer that applies a filter to incoming messages, so that only messages meeting a specific selection criterion are processed.

Figure 7.5. Selective Consumer Pattern



You can implement the selective consumer pattern in FUSE Mediation Router using one of the following approaches:

- "JMS selector" on page 101.
- "JMS selector in ActiveMQ" on page 102
- "Message filter" on page 102.

JMS selector

A JMS selector is a predicate expression involving JMS headers and JMS properties: if the selector evaluates to `true`, the JMS message is allowed to reach the consumer; if the selector evaluates to `false`, the JMS message is blocked. For example, to consume messages from the queue, `selective`, and select only those messages whose country code property is equal to `US`, you could use the following Java DSL route:

```
from("jms:selective?selector=" + java.net.URLEncoder.en
code("CountryCode='US'","UTF-8")).
    to("cxf:bean:replica01");
```

Where the selector string, `CountryCode='US'`, must be URL encoded (using UTF-8 characters) in order to avoid trouble with parsing the query options. This example presumes that the JMS property, `CountryCode`, was set by the sender. For more details about JMS selectors, see "JMS selectors" on page 98.

> **Note**
>
> If a selector is applied to a JMS queue, messages that are not selected remain on the queue (and are thus potentially available to other consumers attached to the same queue).

JMS selector in ActiveMQ

You can also define JMS selectors on ActiveMQ endpoints. For example:

```
from("acivemq:selective?selector=" + java.net.URLEncoder.en
code("CountryCode='US'","UTF-8")).
    to("cxf:bean:replica01");
```

For more details, see ActiveMQ: JMS Selectors[6] and ActiveMQ Message Properties[7].

Message filter

If it is not possible to set a selector on the consumer endpoint, you can insert a filter processor into your route instead. For example, you could define a selective consumer that processes only messages with a US country code using Java DSL, as follows:

```
from("seda:a").filter(header("Country
Code").isEqualTo("US")).process(myProcessor);
```

The same route can be defined using XML configuration, as follows:

```
<camelContext id="buildCustomProcessorWithFilter" xmlns="ht
tp://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <filter>
      <xpath>$CountryCode = 'US'</xpath>
      <process ref="#myProcessor"/>
    </filter>
  </route>
</camelContext>
```

For more information about the FUSE Mediation Router filter processor, see Message Filter on page 54.

[6] http://activemq.apache.org/selectors.html
[7] http://activemq.apache.org/activemq-message-properties.html

Warning

Be careful about using a message filter to select messages from a JMS *queue*. When using a filter processor, blocked messages are simply discarded. Hence, if the messages are consumed from a queue (which allows each message to be consumed only once—see "Competing Consumers" on page 95), blocked messages would not be processed at all. This might not be the behavior you want.

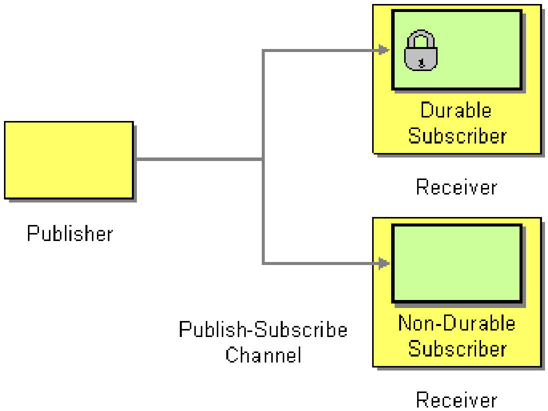
Durable Subscriber

Overview

A *durable subscriber* is a consumer that wants to receive all of the messages sent over a particular publish-subscribe on page 36 channel, including messages sent while the consumer is disconnected from the messaging system. This requires the messaging system to store messages for later replay to the disconnected consumer. There also has to be a mechanism for a consumer to indicate that it wants to establish a durable subscription. Generally, a publish-subscribe channel (or topic) can have both durable and non-durable subscribers, which behave as follows:

- A *non-durable subscriber* can have two states: *connected* and *disconnected*. While a non-durable subscriber is connected to a topic, it receives all of the topic's messages in real time. While a non-durable subscriber is disconnected from a topic, however, it misses all of the message sent during the period of disconnection.
- A *durable subscriber* can have the following states: connected and *inactive*. The inactive state means that the durable subscriber is disconnected from the topic, but wants to receive the messages that arrive in the interim. When the durable subscriber reconnects to the topic, it receives a replay of all the messages sent while it was inactive.

Figure 7.6. Durable Subscriber Pattern



JMS durable subscriber

The JMS component implements the durable subscriber pattern. In order to set up a durable subscription on a JMS endpoint, you need to specify a *client ID*, which identifies this particular connection, and a *durable subscription name*, which identifies the durable subscriber. For example, the following route sets up a durable subscription to the JMS topic, `news`, with a client ID of `conn01` and a durable subscription name of `John.Doe`:

```
from("jms:topic:news?clientId=conn01&durableSubscription
Name=John.Doe").
    to("cxf:bean:newsprocessor");
```

You can also set up a durable subscription using the ActiveMQ endpoint:

```
from("activemq:topic:news?clientId=conn01&durableSubscription
Name=John.Doe").
    to("cxf:bean:newsprocessor");
```

If you want to process the incoming messages concurrently, you could use a SEDA endpoint to fan out the route into multiple, parallel segments, as follows:

```
from("jms:topic:news?clientId=conn01&durableSubscription
Name=John.Doe").
    to("seda:fanout");

from("seda:fanout").to("cxf:bean:newsproc01");
from("seda:fanout").to("cxf:bean:newsproc02");
from("seda:fanout").to("cxf:bean:newsproc03");
```

Where each message is processed only once, because the SEDA component supports the competing consumers pattern.

Idempotent Consumer

Overview

The *idempotent consumer* pattern is used to filter out duplicate messages. For example, consider a scenario where the connection between a messaging system and a consumer endpoint is abruptly lost due to some fault in the system. If the messaging system was in the middle of transmitting a message, it might be unclear whether or not the consumer received the last message. To improve delivery reliability, the messaging system might decide to redeliver such messages as soon as the connection is re-established. Unfortunately, this entails the risk that the consumer might receive duplicate messages and, in some cases, the effect of duplicating a message may have undesirable consequences (such as debiting a sum of money twice from your account). In this scenario, an idempotent consumer could be used to weed out undesired duplicates from the message stream.

Idempotent consumer with in-memory cache

In FUSE Mediation Router, the idempotent consumer pattern is implemented by the `idempotentConsumer()` processor, which takes two arguments:

- `messageIdExpression` — An expression that returns a message ID string for the current message; and
- `messageIdRepository` — A reference to a message ID repository, which stores the IDs of the messages received so far.

As each message comes in, the idempotent consumer processor looks up the current message ID in the repository to see if this message has been seen before. If yes, the message is discarded; if no, the message is allowed to pass and its ID is added to the repository.

For example, the following example uses the `TransactionID` header to filter out duplicates:

```
import static org.apache.camel.processor.idempotent.MemoryMes
sageIdRepository.memoryMessageIdRepository;
...
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a")
          .idempotentConsumer(
             header("TransactionID"),
             memoryMessageIdRepository(200)
          ).to("seda:b");
```

```
    }
};
```

Where the call to `memoryMessageIdRepository(200)` creates an in-memory cache that can hold up to 200 message IDs.

You can also define an idempotent consumer using XML configuration. For example, you can define the preceding route in XML, as follows:

```
<camelContext id="buildIdempotentConsumer" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <idempotentConsumer messageIdRepositoryRef="MsgIDRepos">
      <simple>header.TransactionID</simple>
      <to uri="seda:b"/>
    </idempotentConsumer>
  </route>
</camelContext>

<bean id="MsgIDRepos" class="org.apache.camel.processor.idem
potent.MemoryMessageIdRepository">
    <!-- Specify the in-memory cache size. -->
    <constructor-arg type="int" value="200"/>
</bean>
```

Idempotent consumer with JPA repository

The in-memory cache suffers from the disadvantage that it can easily run out of memory and it does not work in a clustered environment. To avoid these shortcomings, you could use a Java Persistent API (JPA) based repository instead. The JPA message ID repository uses an object-oriented database to store the message IDs. For example, you can define a route that uses a JPA repository for the idempotent consumer, as follows:

```
import org.springframework.orm.jpa.JpaTemplate;

import org.apache.camel.spring.SpringRouteBuilder;
import static org.apache.camel.processor.idempotent.jpa.JpaMes
sageIdRepository.jpaMessageIdRepository;
...
RouteBuilder builder = new SpringRouteBuilder() {
    public void configure() {
        from("seda:a").idempotentConsumer(
          header("TransactionID"),
          jpaMessageIdRepository(bean(JpaTemplate.class),
"myProcessorName")
        ).to("seda:b");
```

```
    }
};
```

Where the JPA message ID repository is initialized with two arguments: a `JpaTemplate` instance, which provides the handle for the JPA database, and a processor name, which uniquely identifies the current idempotent consumer processor. The `SpringRouteBuilder.bean()` method is a shortcut that references a bean defined in the Spring XML file. The `JpaTemplate` bean provides a handle to the underlying JPA database. See the JPA documentation for details of how to configure this bean.

For more details about setting up a JPA repository, see JPA Component[8] documentation, the Spring JPA[9] documentation, and the sample code in the Camel JPA unit test[10].

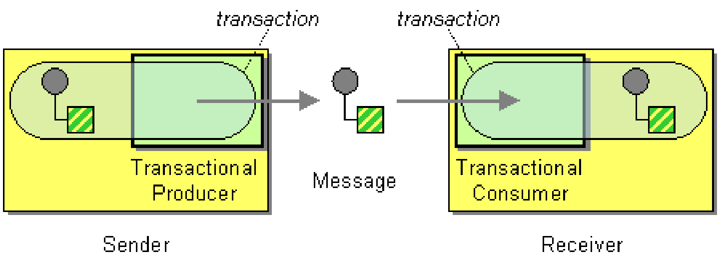
[8] http://activemq.apache.org/camel/jpa.html
[9] http://static.springframework.org/spring/docs/2.5.x/reference/orm.html#orm-jpa
[10] https://svn.apache.org/repos/asf/activemq/camel/trunk/components/camel-jpa/src/test

Transactional Client

Overview

The *transactional client* pattern refers to messaging endpoints that can participate in a transaction. FUSE Mediation Router supports transactions using Spring transaction management[11].

Figure 7.7. Transactional Client Pattern



Transaction oriented endpoints

Not all FUSE Mediation Router endpoints support transactions. Those that do are called *transaction oriented endpoints* (or TOEs). For example, both the JMS component and the ActiveMQ component support transactions.

In order to enable transactions on a component, you need to perform the appropriate initialization before adding the component to the `CamelContext`. For this reason, you need to write some code to initialize your transactional components explicitly.

For example, consider a JMS component that is layered over ActiveMQ. To initialize this as a transactional component, you need to define an instance of `JmsTransactionManager` and an instance of `ActiveMQConnectionFactory`, using the following Spring XML configuration:

```
<bean id="jmsTransactionManager" class="org.springframe
work.jms.connection.JmsTransactionManager">
   <property name="connectionFactory" ref="jmsConnectionFact
ory" />
</bean>

<bean id="jmsConnectionFactory" class="org.apache.activemq.Act
iveMQConnectionFactory">
```

[11] http://static.springframework.org/spring/docs/2.5.x/reference/transaction.html

```
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

You can then initialize the JMS/ActiveMQ component using the following code:

```
// Java
import org.apache.camel.CamelContext;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.spring.SpringRouteBuilder;
import org.apache.camel.spring.SpringCamelContext;
import org.apache.camel.component.jms.JmsComponent;

import javax.jms.ConnectionFactory;

import org.springframework.context.support.ClassPathXmlApplic
ationContext;
import org.springframework.context.ApplicationContext;
import org.springframework.transaction.PlatformTransactionMan
ager;
...
ApplicationContext spring = new ClassPathXmlApplicationCon
text("org/apache/camel/transaction/spring.xml");
CamelContext camelContext = SpringCamelContext.springCamelCon
text(spring);

PlatformTransactionManager transactionManager = (PlatformTrans
actionManager) spring.getBean("jmsTransactionManager");
ConnectionFactory connectionFactory = (ConnectionFactory)
spring.getBean("jmsConnectionFactory");
JmsComponent component = JmsComponent.jmsComponentTrans
acted(connectionFactory, transactionManager);
component.getConfiguration().setConcurrentConsumers(1);
camelContext.addComponent("activemq", component);
```

Transaction propagation policies

Outbound endpoints will automatically enlist in the current transaction context. But what if you do not want your outbound endpoint to enlist in the same transaction as your inbound endpoint? The solution is to add a *transaction policy* to the processing route. First, define the transaction policies in your XML configuration. For example, you can define the transaction policies, `PROPAGATION_REQUIRED`, `PROPAGATION_NOT_SUPPORTED`, and `PROPAGATION_REQUIRES_NEW`, as follows:

```
  <bean id="PROPAGATION_REQUIRED" class="org.springframe
work.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="jmsTransaction
Manager"/>
```

```
  </bean>

  <bean id="PROPAGATION_NOT_SUPPORTED" class="org.springframe
work.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="jmsTransaction
Manager"/>
    <property name="propagationBehaviorName" value="PROPAGA
TION_NOT_SUPPORTED"/>
  </bean>

  <bean id="PROPAGATION_REQUIRES_NEW" class="org.springframe
work.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="jmsTransaction
Manager"/>
    <property name="propagationBehaviorName" value="PROPAGA
TION_REQUIRES_NEW"/>
  </bean>
```

In your `SpringRouteBuilder` class, you need to create new `SpringTransactionPolicy` objects for each of the templates. For example:

```
// Java
public MyRouteBuilder extends SpringRouteBuilder {
  public void configure() {
    ...
    Policy required = new SpringTransactionPolicy(bean(Trans
actionTemplate.class, "PROPAGATION_REQUIRED"));
    Policy notsupported = new SpringTransaction
Policy(bean(TransactionTemplate.class, "PROPAGATION_NOT_SUP
PORTED"));
   Policy requirenew = new SpringTransactionPolicy(bean(Trans
actionTemplate.class, "PROPAGATION_REQUIRES_NEW"));
    ...
  }
}
```

Note

The `org.apache.camel.spring.SpringRouteBuilder` class is a special implementation of the `RouteBuilder` class provided by the FUSE Mediation Router Spring component. It is required for any routes that use Spring transactions. The `SpringRouteBuilder.bean()` method provides a shortcut for looking up bean references in the Spring configuration file.

You can then use the transaction policy objects in your route definitions, as follows:

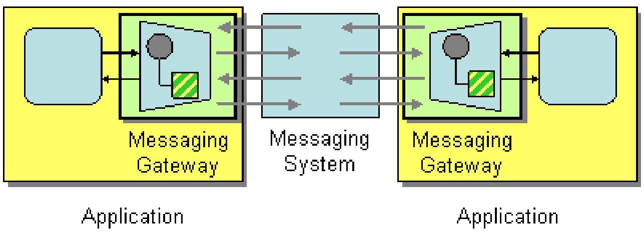
```
// Send to bar in a new transaction
from("activemq:queue:foo").policy(requirenew).to("act
ivemq:queue:bar");

// Send to bar without a transaction.
from("activemq:queue:foo").policy(notsupported ).to("act
ivemq:queue:bar");
```

Messaging Gateway

Overview

The *messaging gateway* pattern describes an approach to integrating with a messaging system, where the messaging system's API remains hidden from the programmer at the application level. In particular, the most common example is where you want to translate synchronous method calls into request/reply message exchanges, without the programmer being aware of this.

Figure 7.8. Messaging Gateway Pattern



The following FUSE Mediation Router components provide this kind of integration with the messaging system:

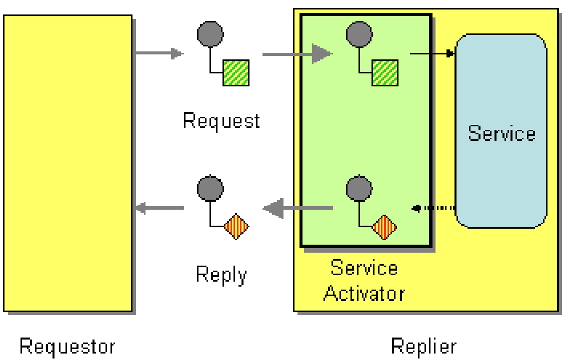
- ????.
- ????.

Service Activator

Overview

The *service activator* pattern describes the scenario where a service's operations are invoked in response to an incoming request message. The service activator is responsible for identifying which operation to call and for extracting the data to use as the operation's parameters. Finally, the service activator invokes an operation using the data extracted from the message. The operation invocation can either be oneway (request only) or two-way (request/reply).

Figure 7.9. Service Activator Pattern



In many respects, a service activator resembles a conventional remote procedure call (RPC), where operation invocations are encoded as messages. The main difference is that a service activator needs to be more flexible. Whereas an RPC framework standardises the request and reply message encodings (for example, Web service operations are encoded as SOAP messages), a service activator typically needs to improvise the mapping between the messaging system and the service's operations.

Bean integration

The main mechanism that FUSE Mediation Router provides to support the service activator pattern is *bean integration*. Bean integration[12] provides a general framework for mapping incoming messages to method invocations on Java objects. For example, the Java fluent DSL provides the processors, `bean()` and `beanRef()`, that you can insert into a route in order to invoke methods on a registered Java bean. The detailed mapping of message data

[12] http://activemq.apache.org/camel/bean-integration.html

to Java method parameters is determined by the *bean binding*, which can be implemented by adding annotations to the bean class.

For example, consider the following route which calls the Java method, `BankBean.getUserAccBalance()`, in order to service requests incoming on a JMS/ActiveMQ queue:

```
from("activemq:BalanceQueries")
  .setProperty("userid", xpath("/Account/Bal
anceQuery/UserID").stringResult())
  .beanRef("bankBean", "getUserAccBalance")
  .to("velocity:file:src/scripts/acc_balance.vm")
  .to("activemq:BalanceResults");
```

The messages pulled from the ActiveMQ endpoint, `activemq:BalanceQueries`, have a simple XML format that provides the user ID of a bank account—for example:

```
<?xml version='1.0' encoding='UTF-8'?>
<Account>
  <BalanceQuery>
    <UserID>James.Strachan</UserID>
  </BalanceQuery>
</Account>
```

The first processor in the route, `setProperty()`, extracts the user ID from the *In* message and stores it in the `userid` exchange property, (this is preferable to storing it in a header, because the *In* headers cease to be available after invoking the bean).

The service activation step is performed by the `beanRef()` processor, which binds the incoming message to the `getUserAccBalance()` method on the Java object identified by the `bankBean` bean ID. The following code shows a sample implementation of the `BankBean` class:

```
// Java
package tutorial;

import org.apache.camel.language.XPath;

public class BankBean {
    public int getUserAccBalance(@XPath("/Account/Bal
anceQuery/UserID") String user) {
        if (user.equals("James.Strachan")) {
            return 1200;
        }
        else {
            return 0;
```

```
        }
      }
}
```

Where the binding of message data to method parameter is enabled by the `@XPath` annotation, which injects the content of the `UserID` XML element into the `user` method parameter. On completion of the call, the return value is inserted into the body of the *Out* message (which is then copied into the *In* message for the next step in the route). In order for the bean to be accessible to the `beanRef()` processor, you must instantiate an instance in Spring XML. For example, you can add the following lines to `META-INF/spring/camel-context.xml` configuration file to instantiate the bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
  ...
  <bean id="bankBean" class="tutorial.BankBean"/>
</beans>
```

Where the bean ID, `bankBean`, identifes this bean instance in the registry.

The output of the bean invocation is fed into a Velocity template, in order to produce a properly formatted result message. The Velocity endpoint, `velocity:file:src/scripts/acc_balance.vm`, specifies the location of a velocity script, which has the following contents:

```
<?xml version='1.0' encoding='UTF-8'?>
<Account>
  <BalanceResult>
    <UserID>${exchange.getProperty("userid")}</UserID>
    <Balance>${body}</Balance>
  </BalanceResult>
</Account>
```

The exchange instance is available as the Velocity variable, `exchange`, which enables you to retrieve the `userid` exchange property, using `${exchange.getProperty("userid")}`. The body of the current *In* message, `${body}`, contains the result of the `getUserAccBalance()` method invocation.

Chapter 8. System Management

The system management patterns describe how to monitor, test, and administer a messaging system.

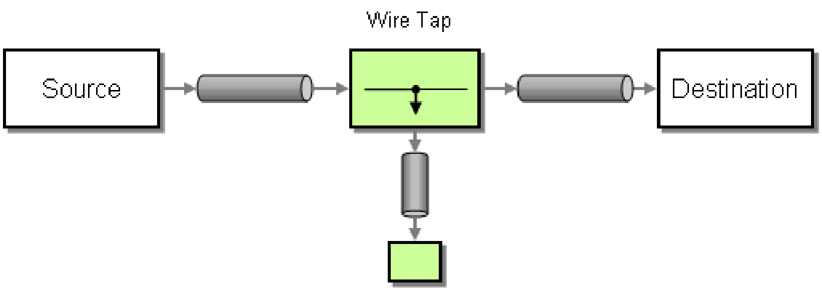
Wire Tap 120

Wire Tap

Overview

The *wire tap* pattern enables you to monitor the messages passing through a channel by duplicating the message stream: one copy of the stream is forwarded to the main channel and another copy of the stream is forwarded to the *tap* endpoint, which monitors the stream.

Figure 8.1. Wire Tap Pattern



Java DSL example

The following example shows how to route a request from an input `queue:a` endpoint to the wire tap location `queue:tap` before it is received by `queue:b`.

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("seda:a").to("seda:tap", "seda:b");
    }
};
```

XML configuration example

The following example shows how to configure the same route in XML:

```
<camelContext id="buildWireTap" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="seda:a"/>
    <to uri="seda:tap"/>
    <to uri="seda:b"/>
  </route>
</camelContext>
```

Appendix A. Migrating from ServiceMix EIP

If you are currently an Apache ServiceMix user, you might already have implemented some Enterprise Integration Patterns using the ServiceMix EIP module. It is recommended that you migrate these legacy patterns to FUSE Mediation Router, which has more extensive support for Enterprise Integration Patterns. After migrating, you can deploy your patterns either into a FUSE ESB container or into a ServiceMix container.

Migrating Endpoints 122
Common Elements 125
ServiceMix EIP Patterns 127
Content-Based Router 129
Content Enricher 131
Message Filter 133
Pipeline 135
Resequencer 137
Static Recipient List 139
Static Routing Slip 141
Wire Tap 143
XPath Splitter 145

Migrating Endpoints

Overview

A typical ServiceMix EIP route exposes a service that consumes exchanges from the NMR. The route also defines one or more target destinations, to which exchanges are sent. In the FUSE Mediation Router environment, the exposed ServiceMix service maps to a *consumer endpoint* and the ServiceMix target destinations map to *producer endpoints*. The FUSE Mediation Router consumer endpoints and producer endpoints are both defined using *endpoint URIs* (see ????).

When migrating endpoints from ServiceMix EIP to FUSE Mediation Router, you will need to express the ServiceMix services/endpoints as FUSE Mediation Router endpoint URIs. You can adopt one of the following approaches:

- Connect to an existing ServiceMix service/endpoint through the ServiceMix Camel module (which integrates FUSE Mediation Router with the NMR).
- Alternatively, if the existing ServiceMix service/endpoint represents a ServiceMix binding component, you could replace the ServiceMix binding component with an equivalent FUSE Mediation Router component (thus bypassing the NMR).

The ServiceMix Camel module

The integration between FUSE Mediation Router and ServiceMix is provided by the `servicemix-camel` module. This module is provided with ServiceMix, but actually implements a plug-in for the FUSE Mediation Router product: the *JBI component* (see ???? and JBI Component[1]).

To access the JBI component from FUSE Mediation Router, make sure that the `servicemix-camel` JAR file is included on your Classpath or, if you are using Maven, include a dependency on the `servicemix-camel` artifact in your project POM. You can then access the JBI component by defining FUSE Mediation Router endpoint URIs with the `jbi:` component prefix.

Translating ServiceMix URIs into FUSE Mediation Router endpoint URIs

ServiceMix defines a flexible format for defining URIs, which is described in detail in ServiceMix URIs[2]. To translate a ServiceMix URI into a FUSE Mediation Router endpoint URI, perform the following steps:

[1] http://activemq.apache.org/camel/jbi.html
[2] http://servicemix.apache.org/uris.html

1. If the ServiceMix URI contains a namespace prefix, replace the prefix by its corresponding namespace.

   For example, after modifying the ServiceMix URI, `service:test:messageFilter`, where `test` corresponds to the namespace, `http://progress.com/demos/test`, you get `service:http://progress.com/demos/test:messageFilter`.

2. Modify the separator character, depending on what kind of namespace appears in the URI:

   - If the namespace starts with `http://`, use the `/` character as the separator between namespace, service name, and endpoint name (if present).

     For example, the URI, `service:http://progress.com/demos/test:messageFilter`, would be modified to `service:http://progress.com/demos/test/messageFilter`.

   - If the namespace starts with `urn:`, use the `:` character as the separator between namespace, service name, and endpoint name (if present).

     For example, `service:urn:progress:com:demos:test:messageFilter`.

3. To create a JBI endpoint URI, add the `jbi:` prefix.

   For example, `jbi:service:http://progress.com/demos/test/messageFilter`.

Example of mapping ServiceMix URIs

For example, consider the following configuration of the static recipient list pattern in ServiceMix EIP. The `eip:exchange-target` elements define some targets using the ServiceMix URI format.

```
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
       xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:test="http://progress.com/demos/test" >
    ...
    <eip:static-recipient-list service="test:recipients" end
point="endpoint">
      <eip:recipients>
        <eip:exchange-target uri="service:test:messageFilter"
 />
```

```
        <eip:exchange-target uri="service:test:trace4" />
      </eip:recipients>
    </eip:static-recipient-list>
    ...
</beans>
```

When the preceding ServiceMix configuration is mapped to an equivalent FUSE Mediation Router configuration, you get the following route:

```
<route>
  <from uri="jbi:endpoint:http://progress.com/demos/test/re
cipients/endpoint"/>
  <to uri="jbi:service:http://progress.com/demos/test/message
Filter"/>
  <to uri="jbi:service:http://progress.com/demos/test/trace4"/>
</route>
```

Replacing ServiceMix bindings with FUSE Mediation Router components

Instead of using the FUSE Mediation Router JBI component to route all your messages through the ServiceMix NMR, you could use one of the many supported FUSE Mediation Router components to connect directly to a consumer or producer endpoint. In particular, when sending messages to an external endpoint, it is frequently more efficient to send the messages directly through a FUSE Mediation Router component rather than sending them through the NMR and a ServiceMix binding.

For details of all the FUSE Mediation Router components that are available, see ???? and FUSE Mediation Router Components[3].

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

Common Elements

Overview

When configuring ServiceMix EIP patterns in a ServiceMix configuration file, there are some common elements that recur in many of the pattern schemas. This section provides a brief overview of these common elements and explains how they can be mapped to equivalent constructs in FUSE Mediation Router.

Exchange target

All of the patterns supported by ServiceMix EIP use the `eip:exchange-target` element to specify JBI target endpoints. Table A.1 on page 125 shows some examples of how to map some sample `eip:exchange-target` elements to FUSE Mediation Router endpoint URIs, where it is assumed that the `test` prefix maps to the `http://progress.com/demos/test` namespace.

Table A.1. Mapping the Exchange Target Element

ServiceMix EIP Target	FUSE Mediation Router Endpoint URI
`<eip:exchange-target interface="HelloWorld" />`	`jbi:interface:HelloWorld`
`<eip:exchange-target service="test:HelloWorldService" />`	`jbi:service:http://progress.com/demos/test/HelloWorldService`
`<eip:exchange-target service="test:HelloWorldService" endpoint="secure" />`	`jbi:service:http://progress.com/demos/test/HelloWorldService/`
`<eip:exchange-target uri="service:test:HelloWorldService" />`	`jbi:service:http://progress.com/demos/test/HelloWorldService`

Predicates

The ServiceMix EIP component lets you define predicate expressions in the XPath language (for example, XPath predicates can appear in `eip:xpath-predicate` elements or in `eip:xpath-splitter` elements, where the XPath predicate is specified using an `xpath` attribute).

ServiceMix XPath predicates can easily be migrated to equivalent constructs in FUSE Mediation Router: that is, either the `xpath` element (in XML configuration) or the `xpath()` command (in Java DSL). For example, the message filter pattern in FUSE Mediation Router can incorporate an XPath predicate as follows:

```
<route>
  <from uri="jbi:endpoint:http://progress.com/demos/test/mes
sageFilter/endpoint">
  <filter>
   <xpath>count(/test:world) = 1</xpath>
   <to uri="jbi:service:http://pro
gress.com/demos/test/trace3"/>
  </filter>
</route>
```

Where the `xpath` element specifies that only messages containing the `test:world` element will pass through the filter.

Note

FUSE Mediation Router also supports a wide range of other scripting languages (such as XQuery, PHP, Python, Ruby, and so on), which can be used to define predicates. For details of all the supported predicate languages, see "Languages for Expressions and Predicates" in *Defining Routes* and "Languages for Expressions and Predicates" in *Defining Routes*.

Namespace contexts

When using XPath predicates in the ServiceMix EIP configuration, it is necessary to define a namespace context using the `eip:namespace-context` element. The namespace can then be referenced using a `namespaceContext` attribute.

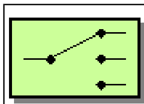
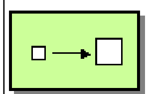
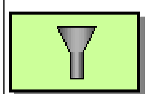
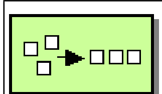
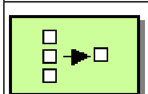
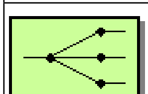
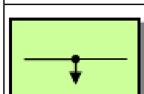
When ServiceMix EIP configuration is migrated to FUSE Mediation Router, however, there is no need to define namespace contexts, because FUSE Mediation Router allows you to define XPath predicates without referencing a namespace context. Hence, you can simply drop the `eip:namespace-context` elements when you migrate to FUSE Mediation Router.

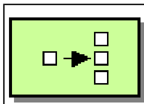
ServiceMix EIP Patterns

Supported patterns

The patterns supported by ServiceMix EIP are shown in Table A.2 on page 127.

Table A.2. ServiceMix EIP Patterns

	Content-Based Router	How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?
	Content Enricher	How do we communicate with another system if the message originator does not have all the required data items available?
	Message Filter	How can a component avoid receiving uninteresting messages?
	Pipeline	How can we perform complex processing on a message while maintaining independence and flexibility?
	Resequencer	How can we get a stream of related but out-of-sequence messages back into the correct order?
	Split Aggregator	How do we combine the results of individual, but related messages so that they can be processed as a whole?
	Static Recipient List	How do we route a message to a list of specified recipients?
	Static Routing Slip	How do we route a message consecutively through a series of processing steps?
	Wire Tap	How do you inspect messages that travel on a point-to-point channel?

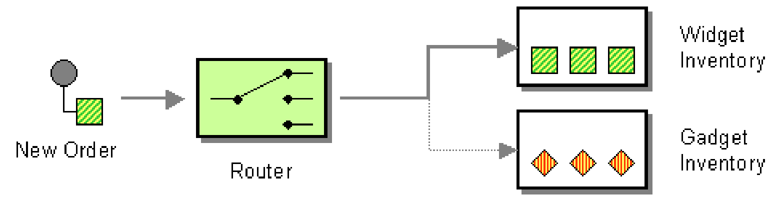
	XPath Splitter	How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?

Content-Based Router

Overview

A *content-based router* enables you to route messages to the appropriate destination, where the routing decision is based on the message contents. This pattern maps to the corresponding content-based router on page 52 pattern in FUSE Mediation Router.

Figure A.1. Content-Based Router Pattern



Example ServiceMix EIP route

The following example shows how to define a content-based router using the ServicMix EIP component. Incoming messages are routed to the `http://test/pipeline/endpoint` endpoint, if a `test:echo` element is present in the message body, and to the `test:recipients` endpoint, otherwise:

```
<eip:content-based-router service="test:router" endpoint="en
dpoint">
  <eip:rules>
    <eip:routing-rule>
      <eip:predicate>
        <eip:xpath-predicate xpath="count(/test:echo) = 1"
namespaceContext="#nsContext" />
      </eip:predicate>
      <eip:target>
        <eip:exchange-target uri="endpoint:test:pipeline:end
point" />
      </eip:target>
    </eip:routing-rule>
    <eip:routing-rule>
      <!-- There is no predicate, so this is the default des
tination -->
      <eip:target>
        <eip:exchange-target service="test:recipients" />
      </eip:target>
    </eip:routing-rule>
```

```
  </eip:rules>
</eip:content-based-router>
```

Equivalent FUSE Mediation Router XML route

The following example shows how to define an equivalent route using FUSE Mediation Router XML configuration:

```
<route>
  <from uri="jbi:endpoint:http://pro
gress.com/demos/test/router/endpoint"/>
  <choice>
    <when>
      <xpath>count(/test:echo) = 1</xpath>
      <to uri="jbi:endpoint:http://pro
gress.com/demos/test/pipeline/endpoint"/>
    </when>
    <otherwise>
      <!-- This is the default destination -->
      <to uri="jbi:service:http://progress.com/demos/test/re
cipients"/>
    </otherwise>
  </choice>
</route>
```

Equivalent FUSE Mediation Router Java DSL route

The following example shows how to define an equivalent route using the FUSE Mediation Router Java DSL:

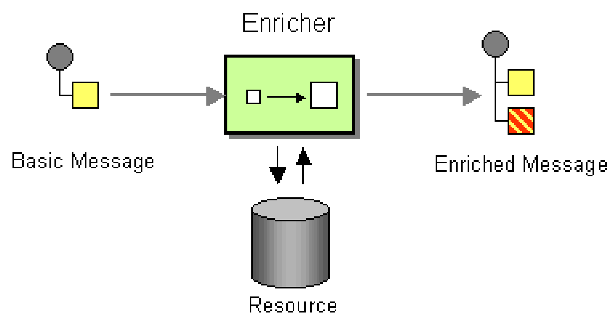
```
from("jbi:endpoint:http://progress.com/demos/test/router/end
point").
    choice().when(xpath("count(/test:echo) = 1")).to("jbi:en
dpoint:http://progress.com/demos/test/pipeline/endpoint").
    otherwise().to("jbi:service:http://pro
gress.com/demos/test/recipients");
```

Content Enricher

Overview

A *content enricher* is a pattern for augmenting a message with missing information. The ServiceMix EIP content enricher is more or less equivalent to a pipeline that adds missing data as the message passes through an enricher target. Consequently, when migrating to FUSE Mediation Router, you can re-implement the ServiceMix content enricher as a FUSE Mediation Router pipeline.

Figure A.2. Content Enricher Pattern



Example ServiceMix EIP route

The following example shows how to define a content enricher using the ServiceMix EIP component. Incoming messages pass through the enricher target, `test:additionalInformationExtracter`, which adds some missing data to the message before the message is sent on to its ultimate destination, `test:myTarget`.

```
<eip:content-enricher service="test:contentEnricher" end
point="endpoint">
  <eip:enricherTarget>
   <eip:exchange-target service="test:additionalInformationEx
tracter" />
  </eip:enricherTarget>
  <eip:target>
    <eip:exchange-target service="test:myTarget" />
```

```
  </eip:target>
</eip:content-enricher>
```

Equivalent FUSE Mediation Router XML route

The following example shows how to define an equivalent route using FUSE Mediation Router XML configuration:

```
<route>
  <from uri="jbi:endpoint:http://progress.com/demos/test/con
tentEnricher/endpoint"/>
  <to uri="jbi:service:http://progress.com/demos/test/addition
alInformationExtracter"/>
  <to uri="jbi:service:http://progress.com/demos/test/myTar
get"/>
</route>
```

Equivalent FUSE Mediation Router Java DSL route

The following example shows how to define an equivalent route using the FUSE Mediation Router Java DSL:

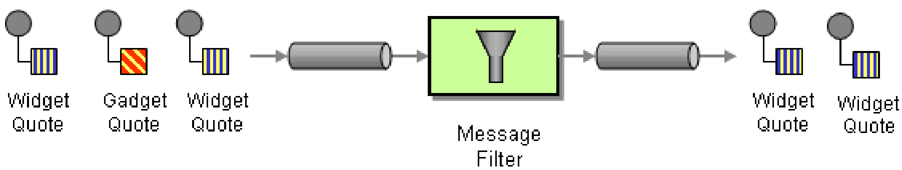
```
from("jbi:endpoint:http://progress.com/demos/test/contentEn
richer/endpoint").
   to("jbi:service:http://progress.com/demos/test/additional
InformationExtracter").
   to("jbi:service:http://progress.com/demos/test/myTarget");
```

Message Filter

Overview

A *message filter* is a processor that eliminates undesired messages based on specific criteria. Filtering is controlled by specifying a predicate in the filter: when the predicate is `true`, the incoming message is allowed to pass; otherwise, it is blocked. This pattern maps to the corresponding message filter on page 54 pattern in FUSE Mediation Router.

Figure A.3. Message Filter Pattern



Example ServiceMix EIP route

The following example shows how to define a message filter using the ServiceMix EIP component. Incoming messages are passed through a filter mechanism that blocks messages that lack a `test:world` element.

```
<eip:message-filter service="test:messageFilter" endpoint="en
dpoint">
  <eip:target>
    <eip:exchange-target service="test:trace3" />
  </eip:target>
  <eip:filter>
    <eip:xpath-predicate xpath="count(/test:world) = 1"
namespaceContext="#nsContext"/>
  </eip:filter>
</eip:message-filter>
```

Equivalent FUSE Mediation Router XML route

The following example shows how to define an equivalent route using FUSE Mediation Router XML configuration:

```
<route>
  <from uri="jbi:endpoint:http://progress.com/demos/test/mes
sageFilter/endpoint">
  <filter>
   <xpath>count(/test:world) = 1</xpath>
   <to uri="jbi:service:http://pro
gress.com/demos/test/trace3"/>
```

```
  </filter>
</route>
```

Equivalent FUSE Mediation Router Java DSL route

The following example shows how to define an equivalent route using the FUSE Mediation Router Java DSL:

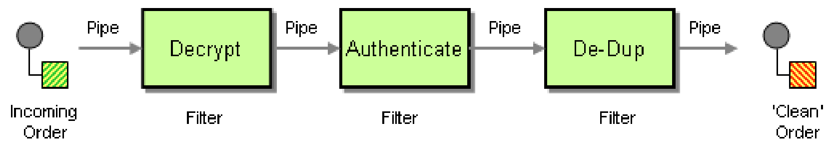
```
from("jbi:endpoint:http://progress.com/demos/test/messageFil
ter/endpoint").
  filter(xpath("count(/test:world) = 1")).
  to("jbi:service:http://progress.com/demos/test/trace3");
```

Pipeline

Overview

The ServiceMix EIP *pipeline* pattern is used to pass messages through a single transformer endpoint, where the transformer's input is taken from the source endpoint and the transformer's output is routed to the target endpoint. This pattern is thus a special case of the more general FUSE Mediation Router pipes and filters on page 26 pattern, which enables you to pass an *In* message through *multiple* transformer endpoints.

Figure A.4. Pipes and Filters Pattern



Example ServiceMix EIP route

The following example shows how to define a pipeline using the ServiceMix EIP component. Incoming messages are passed into the transformer endpoint, `test:decrypt`, and the output from the transformer endpoint is then passed into the target endpoint, `test:plaintextOrder`.

```
<eip:pipeline service="test:pipeline" endpoint="endpoint">
  <eip:transformer>
    <eip:exchange-target service="test:decrypt" />
  </eip:transformer>
  <eip:target>
    <eip:exchange-target service="test:plaintextOrder" />
  </eip:target>
</eip:pipeline>
```

Equivalent FUSE Mediation Router XML route

The following example shows how to define an equivalent route using FUSE Mediation Router XML configuration:

```
<route>
  <from uri="jbi:endpoint:http://pro
gress.com/demos/test/pipeline/endpoint"/>
  <to uri="jbi:service:http://progress.com/demos/test/de
crypt"/>
  <to uri="jbi:service:http://progress.com/demos/test/plain
```

```
textOrder"/>
</route>
```

Equivalent FUSE Mediation Router Java DSL route

The following example shows how to define an equivalent route using the FUSE Mediation Router Java DSL:

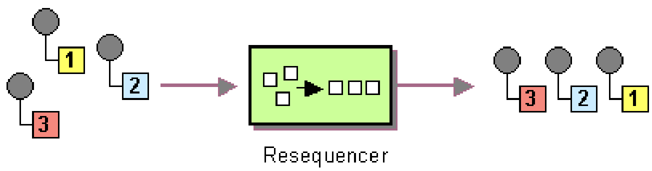
```
from("jbi:endpoint:http://progress.com/demos/test/pipeline/en
dpoint").
    pipeline("jbi:service:http://progress.com/demos/test/de
crypt", "jbi:service:http://progress.com/demos/test/plaintex
tOrder");
```

Resequencer

Overview

The *resequencer* pattern enables you to resequence messages according to the sequence number stored in an NMR property. The ServiceMix EIP resequencer pattern maps to the FUSE Mediation Router resequencer on page 66 configured with the *stream resequencing* algorithm.

Figure A.5. Resequencer Pattern



Sequence number property

The sequence of messages emitted from the resequencer is determined by the value of the sequence number property: messages with a low sequence number are emitted first and messages with a higher number are emitted later. By default, the sequence number is read from the `org.apache.servicemix.eip.sequence.number` property in ServiceMix. But you can customize the name of this property using the `eip:default-comparator` element in ServiceMix.

The equivalent concept in FUSE Mediation Router is a *sequencing expression*, which can be any message-dependent expression. When migrating from ServiceMix EIP, you would normally define an expression that extracts the sequence number from a header (a FUSE Mediation Router header is equivalent to an NMR message property). For example, to extract a sequence number from a `seqnum` header, you could use the simple expression, `header.seqnum`.

Example ServiceMix EIP route

The following example shows how to define a resequencer using the ServiceMix EIP component.

```
<eip:resequencer
  service="sample:Resequencer"
  endpoint="ResequencerEndpoint"
  comparator="#comparator"
  capacity="100"
  timeout="2000">
```

```
  <eip:target>
    <eip:exchange-target service="sample:SampleTarget" />
  </eip:target>
</eip:resequencer>

<!-- Configure default comparator with custom sequence number
 property -->
<eip:default-comparator xml:id="comparator" sequenceNumber
Key="seqnum"/>
```

Equivalent FUSE Mediation Router XML route

The following example shows how to define an equivalent route using FUSE Mediation Router XML configuration:

```
<route>
  <from uri="jbi:endpoint:sample:Resequencer:ResequencerEnd
point"/>
  <resequencer>
    <simple>header.seqnum</simple>
    <to uri="jbi:service:sample:SampleTarget" />
    <stream-config capacity="100" timeout="2000"/>
  </resequencer>
</route>
```

Equivalent FUSE Mediation Router Java DSL route

The following example shows how to define an equivalent route using the FUSE Mediation Router Java DSL:

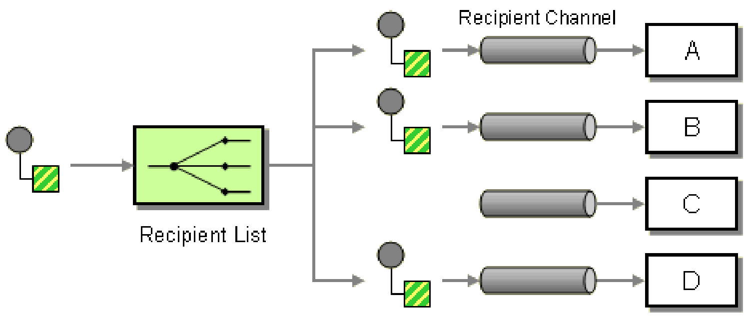
```
from("jbi:endpoint:sample:Resequencer:ResequencerEndpoint").
    resequencer(header("seqnum")).
    stream(new StreamResequencerConfig(100, 2000L)).
    to("jbi:service:sample:SampleTarget");
```

Static Recipient List

Overview

A *recipient list* is a type of router that sends each incoming message to multiple different destinations. The ServiceMix EIP recipient list is restricted to processing *InOnly* and *RobustInOnly* exchange patterns. Moreover, the list of recipients must be static. This pattern maps to the recipient list on page 56 with fixed destination pattern in FUSE Mediation Router.

Figure A.6. Static Recipient List Pattern



Example ServiceMix EIP route

The following example shows how to define a static recipient list using the ServiceMix EIP component. Incoming messages are copied to the `test:messageFilter` endpoint and to the `test:trace4` endpoint.

```
<eip:static-recipient-list service="test:recipients" end
point="endpoint">
  <eip:recipients>
    <eip:exchange-target service="test:messageFilter" />
    <eip:exchange-target service="test:trace4" />
  </eip:recipients>
</eip:static-recipient-list>
```

Equivalent FUSE Mediation Router XML route

The following example shows how to define an equivalent route using FUSE Mediation Router XML configuration:

```
<route>
  <from uri="jbi:endpoint:http://progress.com/demos/test/re
cipients/endpoint"/>
  <to uri="jbi:service:http://progress.com/demos/test/message
```

```
Filter"/>
  <to uri="jbi:service:http://progress.com/demos/test/trace4"/>
</route>
```

Note

The preceding route configuration appears to have the same syntax as a FUSE Mediation Router pipeline pattern. The crucial difference is that the preceding route is intended for processing *InOnly* message exchanges, which are processed in a slightly different way—see "Pipes and Filters" on page 26 for more details.

Equivalent FUSE Mediation Router Java DSL route

The following example shows how to define an equivalent route using the FUSE Mediation Router Java DSL:

```
from("jbi:endpoint:http://progress.com/demos/test/recipients/en
dpoint").
    to("jbi:service:http://progress.com/demos/test/messageFil
ter", "jbi:service:http://progress.com/demos/test/trace4");
```

Static Routing Slip

Overview

The *static routing slip* pattern in the ServiceMix EIP component is used to route an *InOut* message exchange through a series of endpoints. Semantically, it is equivalent to the pipeline on page 26 pattern in FUSE Mediation Router.

Example ServiceMix EIP route

The following example shows how to define a static routing slip using the ServiceMix EIP component. Incoming messages pass through each of the endpoints, `test:procA`, `test:procB`, and `test:procC`, where the output of each endpoint is connected to the input of the next endpoint in the chain. The final endpoint, `tets:procC`, sends its output (*Out* message) back to the caller.

```
<eip:static-routing-slip service="test:routingSlip" end
point="endpoint">
  <eip:targets>
    <eip:exchange-target service="test:procA" />
    <eip:exchange-target service="test:procB" />
    <eip:exchange-target service="test:procC" />
  </eip:targets>
</eip:static-routing-slip>
```

Equivalent FUSE Mediation Router XML route

The following example shows how to define an equivalent route using FUSE Mediation Router XML configuration:

```
<route>
  <from uri="jbi:endpoint:http://progress.com/demos/test/rout
ingSlip/endpoint"/>
  <to uri="jbi:service:http://progress.com/demos/test/procA"/>

  <to uri="jbi:service:http://progress.com/demos/test/procB"/>

  <to uri="jbi:service:http://progress.com/demos/test/procC"/>
</route>
```

Equivalent FUSE Mediation Router Java DSL route

The following example shows how to define an equivalent route using the FUSE Mediation Router Java DSL:

```
from("jbi:endpoint:http://progress.com/demos/test/rout
ingSlip/endpoint").
    pipeline("jbi:service:http://pro
gress.com/demos/test/procA",
```

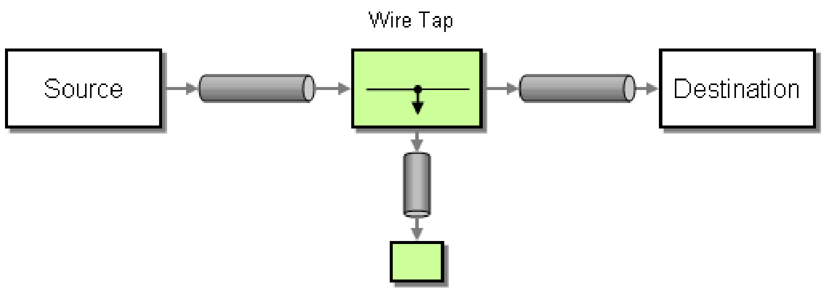
```
             "jbi:service:http://pro
gress.com/demos/test/procB",
             "jbi:service:http://pro
gress.com/demos/test/procC");
```

Wire Tap

Overview

The *wire tap* pattern allows you to route messages to a separate tap location before it is forwarded to the ultimate destination. The ServiceMix EIP wire tap pattern maps to the wire tap on page 120 pattern in FUSE Mediation Router.

Figure A.7. Wire Tap Pattern



Example ServiceMix EIP route

The following example shows how to define a wire tap using the ServiceMix EIP component. The *In* message from the source endpoint is copied to the *In*-listener endpoint, before being forwarded on to the target endpoint. If you want to monitor any returned *Out* messages or *Fault* messages from the target endpoint, you would also need to define an *Out* listener (using the `eip:outListner` element) and a *Fault* listener (using the `eip:faultListener` element).

```
<eip:wire-tap service="test:wireTap" endpoint="endpoint">
  <eip:target>
    <eip:exchange-target service="test:target" />
  </eip:target>
  <eip:inListener>
    <eip:exchange-target service="test:trace1" />
  </eip:inListener>
</eip:wire-tap>
```

Equivalent FUSE Mediation Router XML route

The following example shows how to define an equivalent route using FUSE Mediation Router XML configuration:

```
<route>
  <from uri="jbi:endpoint:http://pro
gress.com/demos/test/wireTap/endpoint"/>
  <to uri="jbi:service:http://progress.com/demos/test/trace1"/>

  <to uri="jbi:service:http://progress.com/demos/test/target"/>
</route>
```

Equivalent FUSE Mediation Router Java DSL route

The following example shows how to define an equivalent route using the FUSE Mediation Router Java DSL:

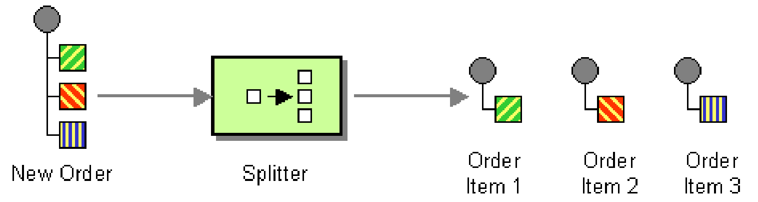
```
from("jbi:endpoint:http://progress.com/demos/test/wireTap/en
dpoint")
  .to("jbi:service:http://progress.com/demos/test/trace1",
      "jbi:service:http://progress.com/demos/test/target");
```

XPath Splitter

Overview

A *splitter* is a type of router that splits an incoming message into a series of outgoing messages, where each of the messages contains a piece of the original message. The ServiceMix EIP XPath splitter pattern is restricted to using the *InOnly* and *RobustInOnly* exchange patterns. The expression that defines how to split up the original message is defined in the XPath language. The XPath splitter pattern maps to the splitter on page 59 pattern in FUSE Mediation Router.

Figure A.8. XPath Splitter Pattern



Forwarding NMR attachments and properties

The `eip:xpath-splitter` element supports a `forwardAttachments` attribute and a `forwardProperties` attribute, either of which can be set to `true`, if you want the splitter to copy the incoming message's attachments or properties to the outgoing messages. The corresponding splitter pattern in FUSE Mediation Router does not support any such attributes. By default, the incoming message's headers are copied to each of the outgoing messages by the FUSE Mediation Router splitter.

Example ServiceMix EIP route

The following example shows how to define a splitter using the ServiceMix EIP component. The specified XPath expression, `/*/*`, would cause an incoming message to split at every occurrence of a nested XML element (for example, the `/foo/bar` and `/foo/car` elements would be split into distinct messages).

```
<eip:xpath-splitter service="test:xpathSplitter" endpoint="en
dpoint"
                    xpath="/*/*" namespaceContext="#nsContext">

  <eip:target>
    <eip:exchange-target uri="service:http://test/router" />
```

```
  </eip:target>
</eip:xpath-splitter>
```

Equivalent FUSE Mediation Router XML route

The following example shows how to define an equivalent route using FUSE Mediation Router XML configuration:

```
<route>
  <from uri="jbi:endpoint:http://progress.com/demos/test/xpath
Splitter/endpoint"/>
  <splitter>
    <xpath>/*/*</xpath>
    <to uri="jbi:service:http://test/router"/>
  </splitter>
</route>
```

Equivalent FUSE Mediation Router Java DSL route

The following example shows how to define an equivalent route using the FUSE Mediation Router Java DSL:

```
from("jbi:endpoint:http://progress.com/demos/test/xpathSplit
ter/endpoint").
    splitter(xpath("/*/*")).to("jbi:service:ht
tp://test/router");
```