Progress FUSE

FUSE™ Mediation Router

Programmer's Guide

Version 1.6
April 2009

PROGRESS SOFTWARE

Programmer's Guide

Version 1.6

Publication date 17 Jul 2009

Legal Notices

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Understanding Message Formats

Before you can begin programming with FUSE Mediation Router, you should have a clear understanding of how messages and message exchanges are modelled. Because FUSE Mediation Router can process many message formats, the basic message type is designed to have an abstract format. FUSE Mediation Router provides the APIs needed to access and transform the data formats that underly message bodies and message headers.

Exchanges

Overview

An *exchange object* is a wrapper that encapsulates a set of related messages provide the primary means of accessing messages in FUSE Mediation Router and they provide the primary means of accessing messages in FUSE Mediation Router. For example, you can access *In*, *Out*, and *Fault* messages using the `getIn()`, `getOut()`, and `getFault()` accessors defined on `Exchange` objects. An important feature of exchanges in FUSE Mediation Router is that they support lazy creation of messages. This can provide a significant optimization in the case of routes that do not require explicit access to messages.

Figure 1.1. Exchange Object Passing through a Route

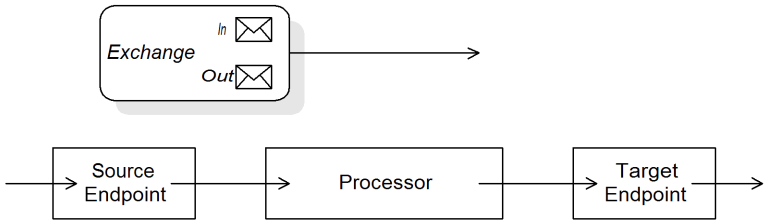


Figure 1.1 on page 12 shows an exchange object passing through a route. In the context of a route, an exchange object gets passed as the argument of the `Processor.process()` method. This means that the exchange object is directly accessible to the source endpoint, the target endpoint, and all of the processors in between.

The Exchange interface

The `org.apache.camel.Exchange` interface defines methods to access *In*, *Out* and *Fault* messages, as shown in Example 1.1 on page 12.

Example 1.1. Exchange Methods

```
Message getIn();
void    setIn(Message in);

Message getOut();
Message getOut(boolean lazyCreate);
void setOut(Message out);

Message getFault();
Message getFault(boolean lazyCreate);
void setFault(Message fault);
```

For a complete description of the methods in the `Exchange` interface, see "The Exchange Interface" on page 108.

Lazy creation of messages

FUSE Mediation Router supports lazy creation of *In*, *Out*, and *Fault* messages. This means that message instances are not created until you try to access them (for example, by calling `getIn()`, `getOut()`, or `getFault()`). The lazy message creation semantics are implemented by the `org.apache.camel.impl.DefaultExchange` class.

If you call one of the no-argument accessors (`getIn()`, `getOut()`, or `getFault()`), or if you call an accessor with the boolean argument equal to `true` (that is, `getIn(true)`, `getOut(true)`, or `getFault(true)`), the default method implementation creates a new message instance, if one does not already exist.

If you call an accessor with the boolean argument equal to `false` (that is, `getIn(false)`, `getOut(false)`, or `getFault(false)`), the default method implementation returns the current message value.[1]

[1] If there is no active method the returned value will be `null`.

Messages

Overview

Message objects represent messages using the following abstract model:

- *Message body*
- *Message headers*
- *Message attachments*

The message body and the message headers can be of arbitrary type (they are declared as type `Object`) and the message attachments are declared to be of type `javax.activation.DataHandler`[2] , which can contain arbitrary MIME types. If you need to obtain a concrete representation of the message contents, you can convert the body and headers to another type using the type converter mechanism and, possibly, using the marshalling and unmarshalling mechanism.

One important feature of FUSE Mediation Router messages is that they support lazy creation of message bodies and headers. In some cases, this means that a message can pass through a route without needing to be parsed at all.

The Message interface

The `org.apache.camel.Message` interface defines methods to access the message body, message headers and message attachments, as shown in Example 1.2 on page 14.

Example 1.2. Message Interface

```
Object getBody();
<T> T getBody(Class<T> type);
void setBody(Object body);
<T> void setBody(Object body, Class<T> type);

Object getHeader(String name);
<T> T getHeader(String name, Class<T> type);
void setHeader(String name, Object value);
Object removeHeader(String name);
Map<String, Object> getHeaders();
void setHeaders(Map<String, Object> headers);

javax.activation.DataHandler getAttachment(String id);
java.util.Map<String, javax.activation.DataHandler> getAttach
```

[2] http://java.sun.com/javaee/5/docs/api/javax/activation/DataHandler.html

```
ments();
java.util.Set<String> getAttachmentNames();
void addAttachment(String id, javax.activation.DataHandler
content)
```

For a complete description of the methods in the `Message` interface, see "The Message Interface" on page 118.

Lazy creation of bodies, headers, and attachments

FUSE Mediation Router supports lazy creation of bodies, headers, and attachments. This means that the objects that represent a message body, a message header, or a message attachment are not created until they are needed.

For example, consider the following route that accesses the `foo` message header from the *In* message:

```
from("SourceURL").filter(header("foo").isEqualTo("bar")).to("Tar
getURL");
```

In this route, if we assume that the component referenced by *`SourceURL`* supports lazy creation, the *In* message headers are not actually parsed until the `header("foo")` call is executed. At that point, the underlying message implementation parses the headers and populates the header map. The message *body* is not parsed until you reach the end of the route, at the `to("`*`TargetURL`*`")` call. At that point, the body is converted into the format required for writing it to the target endpoint, *`TargetURL`*.

By waiting until the last possible moment before populating the bodies, headers, and attachments, you can ensure that unnecessary type conversions are avoided. In some cases, you can completely avoid parsing. For example, if a route contains no explicit references to message headers, a message could traverse the route without ever parsing the headers.

Whether or not lazy creation is implemented in practice depends on the underlying component implementation. In general, lazy creation is valuable for those cases where creating a message body, a message header, or a message attachment is expensive. If the body is left in the form of a raw buffer, it is probably not overly expensive, but parsing headers always adds some cost. For details about implementing a message type that supports lazy creation, see "Implementing the Message Interface" on page 121.

Initial message format

The initial format of an *In* message is determined by the source endpoint, and the initial format of an *Out* message is determined by the target endpoint. If lazy creation is supported by the underlying component, the message

remains unparsed until it is accessed explicitly by the application. Most FUSE Mediation Router components create the message body in a relatively raw form—for example, representing it using types such as byte[], `ByteBuffer`, `InputStream`, or `OutputStream`. This ensures that the overhead required for creating the initial message is minimal. Where more elaborate message formats are required components usually rely on *type converters* or *marshalling processors*.

Type converters

It does not matter what the initial format of the message is, because you can easily convert a message from one format to another using the built-in type converters (see "Built-In Type Converters" on page 19). There are various methods in the FUSE Mediation Router API that expose type conversion functionality. For example, the `convertBodyTo(Class type)` method can be inserted into a route to convert the body of an *In* message, as follows:

```
from("SourceURL").convertBodyTo(String.class).to("TargetURL");
```

Where the body of the *In* message is converted to a `java.lang.String`. The following example shows how to append a string to the end of the *In* message body:

```
from("SourceURL").setBody(bodyAs(String.class).append("My Spe
cial Signature")).to("TargetURL");
```

Where the message body is converted to a string format before appending a string to the end. It is not necessary to convert the message body explicitly in this example. You can also use:

```
from("SourceURL").setBody(body().append("My Special Signa
ture")).to("TargetURL");
```

Where the `append()` method automatically converts the message body to a string before appending its argument.

Type conversion methods in Message

The `org.apache.camel.Message` interface exposes some methods that perform type conversion explicitly:

- `getBody(Class<T> type)`—Returns the message body as type, `T`.
- `getHeader(String name, Class<T> type)`—Returns the named header value as type, `T`.

For the complete list of supported conversion types, see "Built-In Type Converters" on page 19.

Converting to XML

In addition to supporting conversion between simple types (such as byte[], `ByteBuffer`, `String`, and so on), the built-in type converter also supports conversion to XML formats. For example, you can convert a message body to the `org.w3c.dom.Document` type. This conversion is more expensive than the simple conversions, because it involves parsing the entire message and then creating a tree of nodes to represent the XML document structure. You can convert to the following XML document types:

- `org.w3c.dom.Document`
- `javax.xml.transform.sax.SAXSource`

XML type conversions have narrower applicability than the simpler conversions. Because not every message body conforms to an XML structure, you have to remember that this type conversion might fail. On the other hand, there are many scenarios where a router deals exclusively with XML message types.

Marshalling and unmarshalling

Marshalling involves converting a high-level format to a low-level format, and *unmarshalling* involves converting a low-level format to a high-level format. The following two processors are used to perform marshalling or unmarshalling in a route:

- `marshal()`
- `unmarshal()`

For example, to read a serialized Java object from a file and unmarshal it into a Java object, you could use the route definition shown in Example 1.3 on page 17.

Example 1.3. Unmarshalling a Java Object

```
from("file://tmp/appfiles/serialized").unmarshal().
serialization().<FurtherProcessing>.to("TargetURL");
```

For details of how to marshal and unmarshal various data formats, see "Transforming Message Content" in *Defining Routes*.

Final message format

When an *In* message reaches the end of a route, the target endpoint must be able to convert the message body into a format that can be written to the physical endpoint. The same rule applies to *Out* messages that arrive back at the source endpoint. This conversion is usually performed implicitly, using the FUSE Mediation Router type converter. Typically, this involves converting from a low-level format to another low-level format, such as converting from a byte[] array to an `InputStream` type.

Built-In Type Converters

Overview

This section describes the conversions supported by the master type converter. These conversions are built into the FUSE Mediation Router core.

Usually, the type converter is called through convenience functions, such as `Message.getBody(Class<T> type)` or `Message.getHeader(String name, Class<T> type)`. It is also possible to invoke the master type converter directly. For example, if you have an exchange object, `exchange`, you could convert a given value to a `String` as shown in Example 1.4 on page 19.

Example 1.4. Converting a Value to a String

```
org.apache.camel.TypeConverter tc = exchange.getContext().get
TypeConverter();
String str_value = tc.convertTo(String.class, value);
```

Basic type converters

FUSE Mediation Router provides built-in type converters that perform conversions to and from the following basic types:

- `java.io.File`
- `String`
- `byte[]` and `java.nio.ByteBuffer`
- `java.io.InputStream` and `java.io.OutputStream`
- `java.io.Reader` and `java.io.Writer`
- `java.io.BufferedReader` and `java.io.BufferedWriter`
- `java.io.StringReader`

However, not all of these conversions are supported. The built-in converter is mainly focused on providing conversions from the `File` and `String` types. The `File` type can be converted to any of the preceding types, accept `Reader`, `Writer`, and `StringReader`. The `String` type can be converted to `File`, `byte[]`, `ByteBuffer`, `InputStream`, or `StringReader`. The conversion

from `String` to `File` works by interpreting the string as a file name. The trio of `String`, `byte[]`, and `ByteBuffer` are completely inter-convertible.

Note

You can explicitly specify which character encoding to use for conversion from `byte[]` to `String` and from `String` to `byte[]` by setting the `Exchange.CHARSET_NAME` exchange property in the current exchange. For example, to perform conversions using the UTF-8 character encoding, call `exchange.setProperty("Exchange.CHARSET_NAME", "UTF-8")`. The supported character sets are described in the `java.nio.charset.Charset`[3] class.

Collection type converters

FUSE Mediation Router provides built-in type converters that perform conversions to and from the following collection types:

- `Object[]`
- `java.util.Set`
- `java.util.List`

All permutations of conversions between the preceding collection types are supported.

Map type converters

FUSE Mediation Router provides built-in type converters that perform conversions to and from the following map types:

- `java.util.Map`
- `java.util.HashMap`
- `java.util.Hashtable`
- `java.util.Properties`

[3] http://java.sun.com/j2se/1.5.0/docs/api/java/nio/charset/Charset.html

The preceding map types can also be converted into a set, of `java.util.Set` type, where the set elements are of the `MapEntry<K,V>` type.

DOM type converters

You can perform type conversions to the following Document Object Model (DOM) types:

- `org.w3c.dom.Document`—convertible from `byte[]`, `String`, `java.io.File`, and `java.io.InputStream`.
- `org.w3c.dom.Node`
- `javax.xml.transform.dom.DOMSource`—convertible from `String`.
- `javax.xml.transform.Source`—convertible from `byte[]` and `String`.

All permutations of conversions between the preceding DOM types are supported.

SAX type converters

You can also perform conversions to the `javax.xml.transform.sax.SAXSource` type, which supports the SAX event-driven XML parser (see the SAX Web site[4] for details). You can convert to `SAXSource` from the following types:

- `String`
- `InputStream`
- `Source`
- `StreamSource`
- `DOMSource`

Custom type converters

FUSE Mediation Router also enables you to implement your own custom type converters. For details on how to implement a custom type converter, see "Type Converters" on page 35.

[4] http://www.saxproject.org/

Chapter 2. Implementing a Processor

FUSE Mediation Router allows you to implement a custom processor. You can then insert the custom processor into a route to perform operations on exchange objects as they pass through the route.

Processing Models

Overview

Before implementing a processor, you need to consider how the processor will fit into a FUSE Mediation Router route. There are two basic ways to create routes:

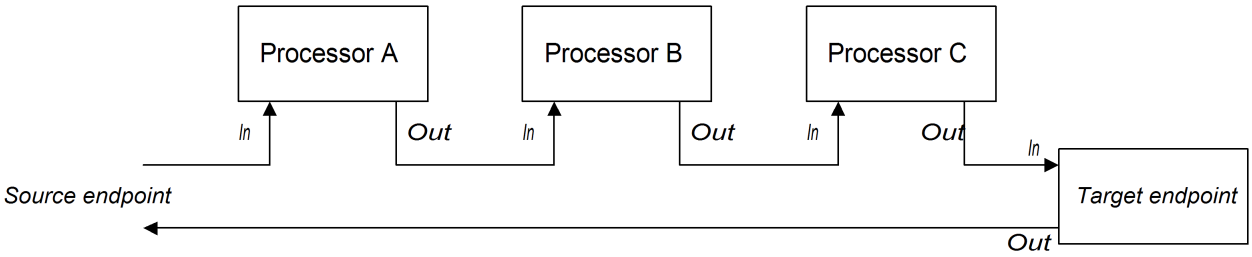
- "Pipelining model"
- "Interceptor chaining"

Pipelining is the preferred method for constructing routes. However, you can accomplish the same routing functionality using either method.

Pipelining model

The *pipelining model* describes the way in which processors are arranged in "Pipes and Filters" in *Implementing Enterprise Integration Patterns*. Pipelining is the most common way to process a sequence of endpoints (a producer endpoint is just a special type of processor). When the processors are arranged in this way, the exchange's *In* and *Out* messages are processed as shown in Figure 2.1 on page 24.

Figure 2.1. Pipelining Model



The processors in the pipeline look like services, where the *In* message is analogous to a request, and the *Out* message is analogous to a reply. In fact, in a realistic pipeline, the nodes in the pipeline are often implemented by Web service endpoints, such as the CXF component.

For example, Example 2.1 on page 25 shows a Java DSL pipeline constructed from a sequence of two processors, `ProcessorA`, `ProcessorB`, and a producer endpoint, *`TargetURI`*.

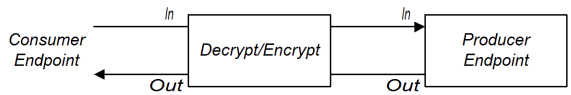
Example 2.1. Java DSL Pipeline

```
from(SourceURI).pipeline(ProcessorA, ProcessorB, TargetURI);
```

Interceptor chaining

An alternative paradigm for linking together the nodes of a route is *interceptor chaining*, where a processor in the route processes the exchange both before *and after* dispatching the exchange to the next processor in the chain. This style of processing is also supported by FUSE Mediation Router, but it is not the usual approach to use. Figure 2.2 on page 25 shows an example of an interceptor processor that implements a custom encryption algorithm.

Figure 2.2. Example of Interceptor Chaining



In this example, incoming messages are encrypted in a custom format. The interceptor first decrypts the *In* message, then dispatches it to the Web services endpoint, `cxf:bean:processTxn`, and finally, the reply (*Out* message) is encrypted using the custom format, before being sent back through the consumer endpoint. Using the interceptor chaining approach, therefore, a single interceptor instance can modify both the request *and* the response.

For example, if you want to define a route with an HTTP port that services incoming requests encoded using custom encryption, you can define a route like the following:

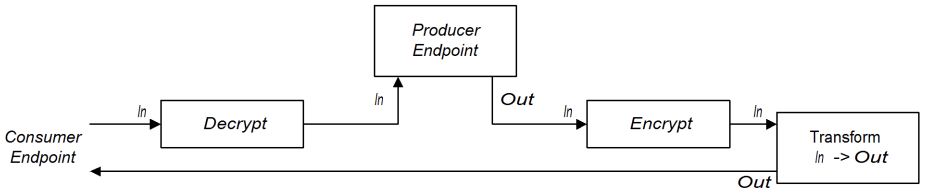
```
from("jetty:http://localhost:8080/foo")
    .intercept(new MyDecryptEncryptInterceptor())
    .to("cxf:bean:processTxn");
```

Where the class, `MyDecryptEncryptInterceptor`, is implemented by inheriting from the class, `org.apache.camel.processor.DelegateProcessor`.

Comparison of pipelining and interceptor chaining

Although it is possible to implement encryption using an interceptor processor, this is not a common way of programming in FUSE Mediation Router. A more typical approach is shown in Figure 2.3 on page 26.

Figure 2.3. Pipeline Alternative to Interceptor Chaining



In this example, the encryption functionality is implemented in a separate processor from the decryption functionality. The resulting processor pipeline is semantically equivalent to the original interceptor chain shown in Figure 2.2 on page 25. One slight complication of the pipeline route is that it requires the addition of a `transform` processor at the end of the route to copy the *In* message to the *Out* message and making the reply message available to the HTTP consumer endpoint. An alternative solution to this problem is to implement the encrypt processor so that it creates an *Out* message directly.

To implement the pipeline approach shown in Figure 2.3 on page 26, you can define a route similar to Example 2.2 on page 26.

Example 2.2. Pipeline Alternative to Encryption Interceptor

```
from("jetty:http://localhost:8080/foo")
    .process(new MyDecryptProcessor())
    .to("cxf:bean:processTxn")
    .process(new MyEncryptProcessor())
    .transform(body());
```

The final processor node, `transform(body())`, has the effect of copying the *In* message to the *Out* message (the *In* message body is copied explicitly and the *In* message headers are copied implicitly).

Implementing a Simple Processor

Overview

This section describes how to implement a simple processor that executes message processing logic before delegating the exchange to the next processor in the route.

Processor interface

Simple processors are created by implementing the `org.apache.camel.Processor` interface. As shown in Example 2.3 on page 27, the interface defines a single method, `process()`, which processes an exchange object.

Example 2.3. Processor Interface

```
package org.apache.camel;

public interface Processor {
    void process(Exchange exchange) throws Exception;
}
```

Implementing the Processor interface

To create a simple processor you must implement the `Processor` interface and provide the logic for the `process()` method. Example 2.4 on page 27 shows the outline of a simple processor implementation.

Example 2.4. Simple Processor Implementation

```
import org.apache.camel.Processor;

public class MyProcessor implements Processor {
    public MyProcessor() { }

    public void process(Exchange exchange) throws Exception
    {
        // Insert code that gets executed *before* delegating

        // to the next processor in the chain.
        ...
    }
}
```

All of the code in the `process()` method gets executed *before* the exchange object is delegated to the next processor in the chain.

For examples of how to access the message body and header values inside a simple processor, see "Accessing Message Content" on page 29.

Inserting the simple processor into a route

Use the `process()` DSL command to insert a simple processor into a route. Create an instance of your custom processor and then pass this instance as an argument to the `process()` method, as follows:

```
org.apache.camel.Processor myProc = new MyProcessor();

from("SourceURL").process(myProc).to("TargetURL");
```

Accessing Message Content

Accessing message headers

Message headers typically contain the most useful message content from the perspective of a router, because headers are often intended to be processed in a router service. To access header data, you must first get the message from the exchange object (for example, using `Exchange.getIn()`), and then use the `Message` interface to retrieve the individual headers (for example, using `Message.getHeader()`).

Example 2.5 on page 29 shows an example of a custom processor that accesses the value of a header named `Authorization`. This example uses the `ExchangeHelper.getMandatoryHeader()` method, which eliminates the need to test for a null header value.

Example 2.5. Accessing an Authorization Header

```
import org.apache.camel.*;
import org.apache.camel.util.ExchangeHelper;

public class MyProcessor implements Processor {
  public void process(Exchange exchange) {
    String auth = ExchangeHelper.getMandatoryHeader(exchange,
 "Authorization", String.class);
    // process the authorization string...
    // ...
  }
}
```

For full details of the `Message` interface, see "Messages" on page 14.

Accessing the message body

You can also access the message body. For example, to append a string to the end of the *In* message, you can use the processor shown in Example 2.6 on page 29.

Example 2.6. Accessing the Message Body

```
import org.apache.camel.*;
import org.apache.camel.util.ExchangeHelper;

public class MyProcessor implements Processor {
    public void process(Exchange exchange) {
        Message in = exchange.getIn();
        in.setBody(in.getBody(String.class) + " World!");
```

```
    }
}
```

Accessing message attachments

You can access a message's attachments using either the `Message.getAttachment()` method or the `Message.getAttachments()` method. See Example 1.2 on page 14 for more details.

The ExchangeHelper Class

Overview

The `org.apache.camel.util.ExchangeHelper`[1] class is a FUSE Mediation Router utility class that provides methods that are useful when implementing a processor.

Resolve an endpoint

The static `resolveEndpoint()` method is one of the most useful methods in the `ExchangeHelper` class. You use it inside a processor to create new `Endpoint` instances on the fly.

Example 2.7. The `resolveEndpoint()` Method

```
public final class ExchangeHelper {
    ...
    @SuppressWarnings({"unchecked" })
   public static <E extends Exchange> Endpoint<E> resolveEnd
point(E exchange, Object value)
        throws NoSuchEndpointException { ... }
    ...
}
```

The first argument to `resolveEndpoint()` is an exchange instance, and the second argument is usually an endpoint URI string. Example 2.8 on page 31 shows how to create a new file endpoint from an exchange instance `exchange`

Example 2.8. Creating a File Endpoint

```
Endpoint file_endp = ExchangeHelper.resolveEndpoint(exchange,
 "file://tmp/messages/in.xml");
```

Wrapping the exchange accessors

The `ExchangeHelper` class provides several static methods of the form `getMandatory`*`BeanProperty`*`()`, which wrap the corresponding `get`*`BeanProperty`*`()` methods on the `Exchange` class. The difference between them is that the original `get`*`BeanProperty`*`()` accessors return `null`, if the corresponding property is unavailable, and the `getMandatory`*`BeanProperty`*`()` wrapper methods throw a Java exception. The following wrapper methods are implemented in the `ExchangeHelper` class:

```
public final class ExchangeHelper {
    ...
```

[1] http://activemq.apache.org/camel/maven/camel-core/apidocs/org/apache/camel/util/ExchangeHelper.html

```
   public static <T> T getMandatoryProperty(Exchange exchange,
 String propertyName, Class<T> type)
        throws NoSuchPropertyException { ... }

   public static <T> T getMandatoryHeader(Exchange exchange,
 String propertyName, Class<T> type)
        throws NoSuchHeaderException { ... }

   public static Object getMandatoryInBody(Exchange exchange)

        throws InvalidPayloadException { ... }

   public static <T> T getMandatoryInBody(Exchange exchange,
 Class<T> type)
        throws InvalidPayloadException { ... }

   public static Object getMandatoryOutBody(Exchange exchange)

        throws InvalidPayloadException { ... }

   public static <T> T getMandatoryOutBody(Exchange exchange,
 Class<T> type)
        throws InvalidPayloadException { ... }
    ...
}
```

Testing the exchange pattern

Several different exchange patterns are compatible with holding an *In* message. Several different exchange patterns are also compatible with holding an *Out* message. To provide a quick way of checking whether or not an exchange object is capable of holding an *In* message or an *Out* message, the `ExchangeHelper` class provides the following methods:

```
public final class ExchangeHelper {
    ...
    public static boolean isInCapable(Exchange exchange) {
... }

    public static boolean isOutCapable(Exchange exchange) {
... }
    ...
}
```

Get the In message's MIME content type

If you want to find out the MIME content type of the exchange's *In* message, you can access it by calling the `ExchangeHelper.getContentType(exchange)` method. To implement

this, the `ExchangeHelper` object looks up the value of the *In* message's `Content-Type` header—this method relies on the underlying component to populate the header value).

Chapter 3. Type Converters

FUSE Mediation Router has a built-in type conversion mechanism, which is used to convert message bodies and message headers to different types. This chapter explains how to extend the type conversion mechanism by adding your own custom converter methods.

Type Converter Architecture

Overview

This section describes the overall architecture of the type converter mechanism, which you must understand, if you want to write custom type converters. If you only need to use the built-in type converters, see "Understanding Message Formats" on page 11.

Type converter interface

Example 3.1 on page 36 shows the definition of the `org.apache.camel.TypeConverter` interface, which all type converters must implement.

Example 3.1. TypeConverter Interface

```
package org.apache.camel;

public interface TypeConverter {
    <T> T convertTo(Class<T> type, Object value);
}
```

Master type converter

The FUSE Mediation Router type converter mechanism follows a master/slave pattern. There are many *slave* type converters, which are each capable of performing a limited number of type conversions, and a single *master* type converter, which aggregates the type conversions performed by the slaves. The master type converter acts as a front-end for the slave type converters. When you request the master to perform a type conversion, it selects the appropriate slave and delegates the conversion task to that slave.

For users of the type conversion mechanism, the master type converter is the most important because it provides the entry point for accessing the conversion mechanism. During start up, FUSE Mediation Router automatically associates a master type converter instance with the `CamelContext` object. To obtain a reference to the master type converter, you call the `CamelContext.getTypeConverter()` method. For example, if you have an exchange object, `exchange`, you can obtain a reference to the master type converter as shown in Example 3.2 on page 37.

Example 3.2. Getting a Master Type Converter

```
org.apache.camel.TypeConverter tc = exchange.getContext().getTypeConverter();
```

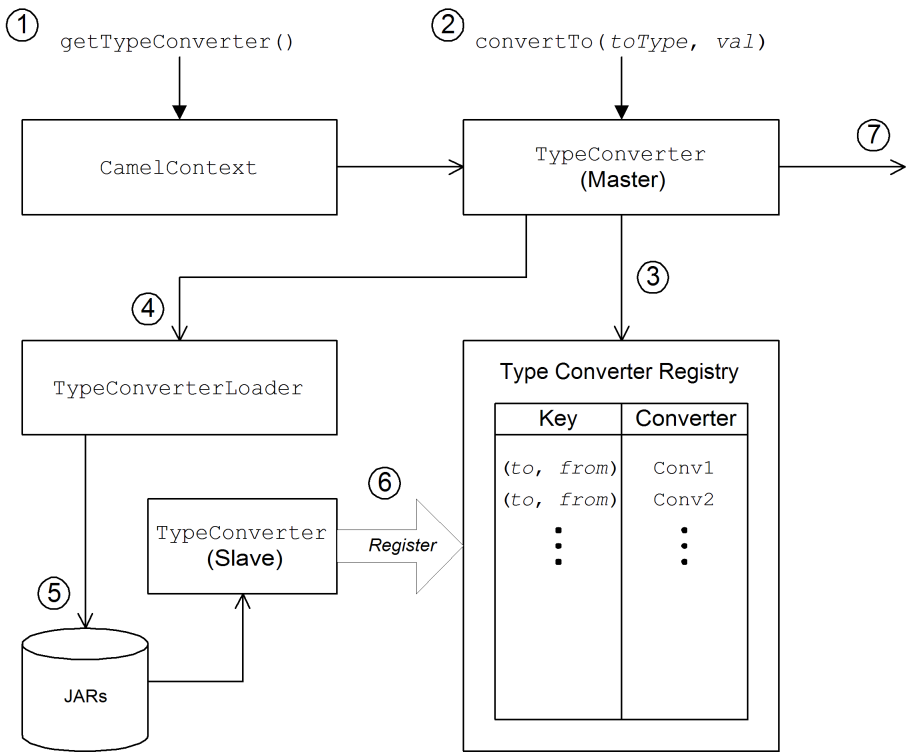
Type converter loader

The master type converter uses a *type converter loader* to populate the registry of slave type converters. A type converter loader is any class that implements the `TypeConverterLoader` interface. FUSE Mediation Router currently uses only one kind of type converter loader—the *annotation type converter loader* (of `AnnotationTypeConverterLoader` type).

Type conversion process

Figure 3.1 on page 37 gives an overview of the type conversion process, showing the steps involved in converting a given data value, `value`, to a specified type, `toType`.

Figure 3.1. Type Conversion Process



The type conversion mechanism proceeds as follows:

1. The `CamelContext` object holds a reference to the master `TypeConverter` instance. The first step in the conversion process is to retrieve the master type converter by calling `CamelContext.getTypeConverter()`.

2. Type conversion is initiated by calling the `convertTo()` method on the master type converter. This method instructs the type converter to convert the data object, `value`, from its original type to the type specified by the `toType` argument.

3. Because the master type converter is a front end for many different slave type converters, it looks up the appropriate slave type converter by checking a registry of type mappings The registry of type converters is keyed by a type mapping pair `(toType, fromType)`. If a suitable type converter is found in the registry, the master type converter calls the slave's `convertTo()` method and returns the result.

4. If a suitable type converter *cannot* be found in the registry, the master type converter loads a new type converter, using the type converter loader.

5. The type converter loader searches the available JAR libraries on the classpath to find a suitable type converter. Currently, the loader strategy that is used is implemented by the annotation type converter loader, which attempts to load a class annotated by the `org.apache.camel.Converter` annotation. See "Create a TypeConverter file" on page 40.

6. If the type converter loader is successful, a new slave type converter is loaded and entered into the type converter registry. This type converter is then used to convert the `value` argument to the `toType` type.

7. If the data is successfully converted, the converted data value is returned. If the conversion does not succeed, `null` is returned.

Implementing a Custom Type Converter

Overview

The type conversion mechanism can easily be customized by adding a new slave type converter. This section describes how to implement a slave type converter and how to integrate it with FUSE Mediation Router, so that it is automatically loaded by the annotation type converter loader.

How to implement a type converter

To implement a custom type converter, perform the following steps:

1. "Implement an annotated converter class" .
2. "Create a TypeConverter file" .
3. "Package the type converter" .

Implement an annotated converter class

You can implement a custom type converter class using the `@Converter` annotation. You must annotate the class itself and each of the methods intended to perform type conversion. Each converter method must take a single argument, which defines the *from* type, and a non-void return value, which defines the *to* type. The type converter loader uses Java reflection to find the annotated methods and integrate them into the type converter mechanism. Example 3.3 on page 39 shows an example of an annotated converter class that defines a single converter method for converting from `java.io.File` to `java.io.InputStream`.

Example 3.3. Example of an Annotated Converter Class

```
package com.YourDomain.YourPackageName;

import org.apache.camel.Converter;

import java.io.*;

@Converter
public class IOConverter {
    private IOConverter() {
    }

    @Converter
   public static InputStream toInputStream(File file) throws
 FileNotFoundException {
        return new BufferedInputStream(new FileInput
```

```
Stream(file));
    }
}
```

The `toInputStream()` method is responsible for performing the conversion from the `File` type to the `InputStream` type.

Note

The method name is unimportant, and can be anything you choose. What is important are the argument type, the return type, and the presence of the `@Converter` annotation.

Create a TypeConverter file

To enable the discovery mechanism (which is implemented by the *annotation type converter loader*) for your custom converter, create a `TypeConverter` file at the following location:

```
META-INF/services/org/apache/camel/TypeConverter
```

The `TypeConverter` file must contain a comma-separated list of package names identifying the packages that contain type converter classes. For example, if you want the type converter loader to search the `com.YourDomain.YourPackageName` package for annotated converter classes, the `TypeConverter` file would have the following contents:

```
com.YourDomain.YourPackageName
```

Package the type converter

The type converter is packaged as a JAR file containing the compiled classes of your custom type converters and the `META-INF` directory. Put this JAR file on your classpath to make it available to your FUSE Mediation Router application.

Chapter 4. Implementing a Component

This chapter provides a general overview of the approaches can be used to implement a FUSE Mediation Router component.

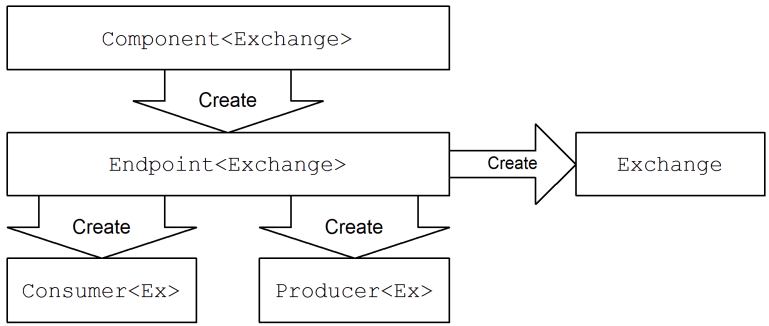
Component Architecture

Factory Patterns for a Component

Overview

A FUSE Mediation Router component consists of a set of classes that are related to each other through a factory pattern. The primary entry point to a component is the `Component` object itself (an instance of `org.apache.camel.Component` type). You can use the `Component` object as a factory to create `Endpoint` objects, which in turn act as factories for creating `Consumer`, `Producer`, and `Exchange` objects. These relationships are summarized in Figure 4.1 on page 43

Figure 4.1. Component Factory Patterns



Component

A component implementation is an endpoint factory. The main task of a component implementor is to implement the `Component.createEndpoint()` method, which is responsible for creating new endpoints on demand.

Each kind of component must be associated with a *component prefix* that appears in an endpoint URI. For example, the file component is usually associated with the `file` prefix, which can be used in an endpoint URI like `file://tmp/messages/input`. When you install a new component in FUSE Mediation Router, you must define the association between a particular component prefix and the name of the class that implements the component.

Endpoint

Each endpoint instance encapsulates a particular endpoint URI. Every time FUSE Mediation Router encounters a new endpoint URI, it creates a new endpoint instance.

Endpoints must implement the `org.apache.camel.Endpoint` interface. The `Endpoint` interface defines the following factory methods:

- `createConsumer()` and `createPollingConsumer()`—Creates a consumer endpoint, which represents the source endpoint at the beginning of a route.
- `createProducer()`—Creates a producer endpoint, which represents the target endpoint at the end of a route.
- `createExchange()`—Creates an exchange object, which encapsulates the messages passed up and down the route.

An endpoint object is also a factory for creating consumer endpoints and producer endpoints.

Consumer

Consumer endpoints *consume* requests. They always appear at the start of a route and they encapsulate the code responsible for receiving incoming requests and dispatching outgoing replies. From a service-oriented prospective a consumer represents a *service*.

Consumers must implement the `org.apache.camel.Consumer` interface. There are a number of different patterns you can follow when implementing a consumer. These patterns are described in "Consumer Patterns and Threading" on page 47.

Producer

Producer endpoints *produce* requests. They always appears at the end of a route and they encapsulate the code responsible for dispatching outgoing requests and receiving incoming replies. From a service-oriented prospective a producer represents a *service consumer*.

Producers must implement the `org.apache.camel.Producer` interface. You can optionally implement the producer to support an asynchronous style of processing. See "Asynchronous Processing" on page 51 for details.

Exchange

Exchange objects encapsulate a related set of messages. For example, one kind of message exchange is a synchronous invocation, which consists of a request message and its related reply.

Exchanges must implement the `org.apache.camel.Exchange` interface. The default implementation, `DefaultExchange`, is sufficient for many component implementations. However, if you want to associated extra data

with the exchanges or have the exchanges preform additional processing, it can be useful to customize the exchange implementation.

Message

There are three different kinds of messages:

- *In* messages
- *Out* messages
- *Fault* messages

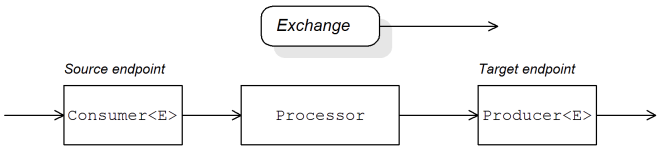
All of the message types are represented by the same Java object, `org.apache.camel.Message`. It is not always necessary to customize the message implementation—the default implementation, `DefaultMessage`, is usually adequate.

Using a Component in a Route

Overview

A FUSE Mediation Router route is essentially a pipeline of processors, of `org.apache.camel.Processor` type. Messages are encapsulated in an exchange object, `E`, which gets passed from node to node by invoking the `process()` method. The architecture of the processor pipeline is illustrated in Figure 4.2 on page 46.

Figure 4.2. Consumer and Producer Instances in a Route



Source endpoint

At the start of the route, you have the source endpoint, which is represented by an `org.apache.camel.Consumer` object. The source endpoint is responsible for accepting incoming request messages and dispatching replies. When constructing the route, FUSE Mediation Router creates the appropriate `Consumer` type based on the component prefix from the endpoint URI, as described in "Factory Patterns for a Component" on page 43.

Processors

Each intermediate node in the pipeline is represented by a processor object (implementing the `org.apache.camel.Processor` interface). You can insert either standard processors (for example, `filter`, `throttler`, or `delayer`) or insert your own custom processor implementations.

Target endpoint

At the end of the route is the target endpoint, which is represented by an `org.apache.camel.Producer` object. Because it comes at the end of a processor pipeline, the producer is also a processor object (implementing the `org.apache.camel.Processor` interface). The target endpoint is responsible for sending outgoing request messages and receiving incoming replies. When constructing the route, FUSE Mediation Router creates the appropriate `Producer` type based on the component prefix from the endpoint URI.

Consumer Patterns and Threading

Overview

The pattern used to implement the consumer determines the threading model used in processing the incoming exchanges. Consumers can be implemented using one of the following patterns:

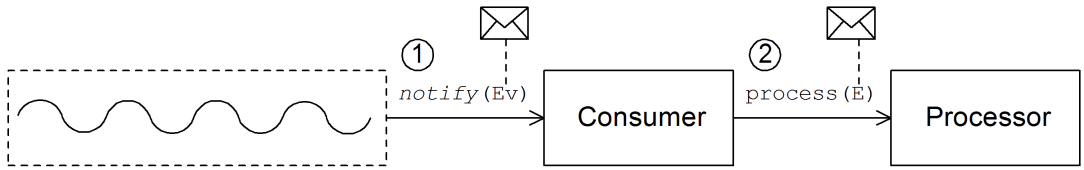
- "Event-driven pattern" —The consumer is driven by an external thread.
- "Scheduled poll pattern" —The consumer is driven by a dedicated thread pool.
- "Polling pattern" —The threading model is left undefined.

Event-driven pattern

In the event-driven pattern, the processing of an incoming request is initiated when another part of the application (typically a third-party library) calls a method implemented by the consumer. A good example of an event-driven consumer is the FUSE Mediation Router JMX component, where events are initiated by the JMX library. The JMX library calls the `handleNotification()` method to initiate request processing—see Example 7.2 on page 92 for details.

Figure 4.3 on page 47 shows an outline of the event-driven consumer pattern. In this example, it is assumed that processing is triggered by a call to the *`notify()`* method.

Figure 4.3. Event-Driven Consumer



The event-driven consumer processes incoming requests as follows:

1. The consumer must implement a method to receive the incoming event (in Figure 4.3 on page 47 this is represented by the *`notify()`* method).

   The thread that calls *`notify()`* is normally a separate part of the application, so the consumer's threading policy is externally driven.

For example, in the case of the JMX consumer implementation, the consumer implements the `NotificationListener.handleNotification()` method to receive notifications from JMX. The threads that drive the consumer processing are created within the JMX layer.

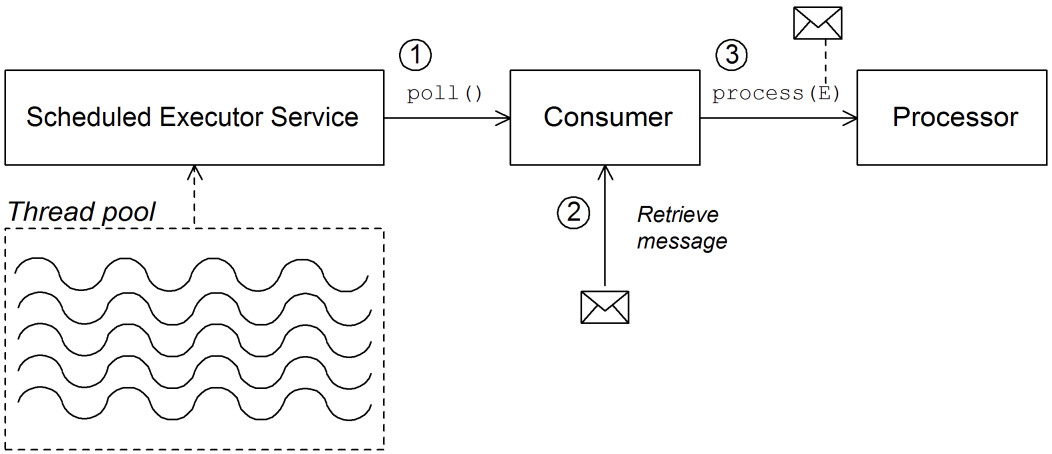
2. In the body of the *`notify()`* method, the consumer first converts the incoming event into an exchange object, `E`, and then calls `process()` on the next processor in the route, passing the exchange object as its argument.

Scheduled poll pattern

In the scheduled poll pattern, the consumer retrieves incoming requests by checking at regular time intervals whether or not a request has arrived. Checking for requests is scheduled automatically by a built-in timer class, the *scheduled executor service*, which is a standard pattern provided by the `java.util.concurrent` library. The scheduled executor service executes a particular task at timed intervals and it also manages a pool of threads, which are used to run the task instances.

Figure 4.4 on page 48 shows an outline of the scheduled poll consumer pattern.

Figure 4.4. Scheduled Poll Consumer



The scheduled poll consumer processes incoming requests as follows:

1. The scheduled executor service has a pool of threads at its disposal, that can be used to initiate consumer processing. After each scheduled time interval has elapsed, the scheduled executor service attempts to grab a free thread from its pool (there are five threads in the pool by default). If a free thread is available, it uses that thread to call the `poll()` method on the consumer.
2. The consumer's `poll()` method is intended to trigger processing of an incoming request. In the body of the `poll()` method, the consumer attempts to retrieve an incoming message. If no request is available, the `poll()` method returns immediately.
3. If a request message is available, the consumer inserts it into an exchange object and then calls `process()` on the next processor in the route, passing the exchange object as its argument.

Polling pattern

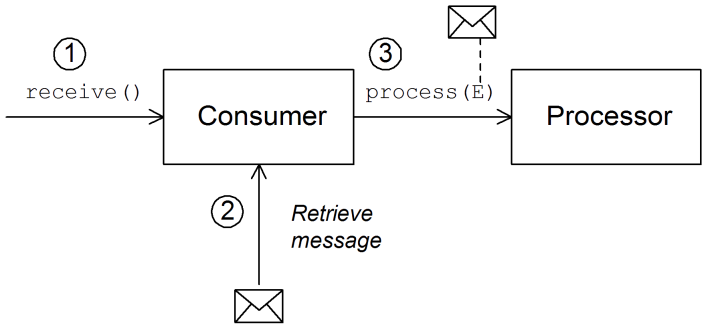
In the polling pattern, processing of an incoming request is initiated when a third-party calls one of the consumer's polling methods:

- `receive()`
- `receiveNoWait()`
- `receive(long timeout)`

It is up to the component implementation to define the precise mechanism for initiating calls on the polling methods. This mechanism is not specified by the polling pattern.

Figure 4.5 on page 50 shows an outline of the polling consumer pattern.

Figure 4.5. Polling Consumer



The polling consumer processes incoming requests as follows:

1. Processing of an incoming request is initiated whenever one of the consumer's polling methods is called. The mechanism for calling these polling methods is implementation defined.

2. In the body of the `receive()` method, the consumer attempts to retrieve an incoming request message. If no message is currently available, the behavior depends on which receive method was called.

   - `receiveNoWait()` returns immediately
   - `receive(long timeout)` waits for the specified timeout interval[1] before returning
   - `receive()` waits until a message is received

3. If a request message is available, the consumer inserts it into an exchange object and then calls `process()` on the next processor in the route, passing the exchange object as its argument.

[1] The timeout interval is typically specified in milliseconds.

Asynchronous Processing

Overview

Producer endpoints normally follow a *synchronous* pattern when processing an exchange. When the preceding processor in a pipeline calls `process()` on a producer, the `process()` method blocks until a reply is received. In this case, the processor's thread remains blocked until the producer has completed the cycle of sending the request and receiving the reply.

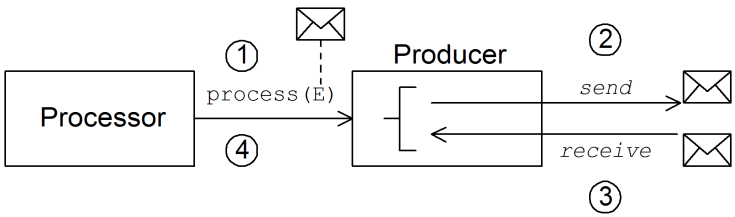
Sometimes, however, you might prefer to decouple the preceding processor from the producer, so that the processor's thread is released immediately and the `process()` call does *not* block. In this case, you should implement the producer using an *asynchronous* pattern, which gives the preceding processor the option of invoking a non-blocking version of the `process()` method.

To give you an overview of the different implementation options, this section describes both the synchronous and the asynchronous patterns for implementing a producer endpoint.

Synchronous producer

Figure 4.6 on page 51 shows an outline of a synchronous producer, where the preceding processor blocks until the producer has finished processing the exchange.

Figure 4.6. Synchronous Producer



The synchronous producer processes an exchange as follows:

1. The preceding processor in the pipeline calls the synchronous `process()` method on the producer to initiate synchronous processing. The synchronous `process()` method takes a single exchange argument.

2. In the body of the `process()` method, the producer sends the request (*In* message) to the endpoint.

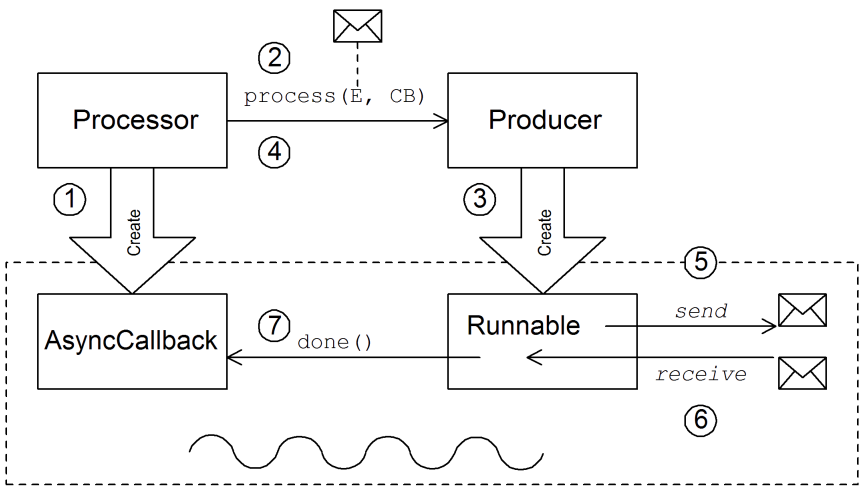
3. If required by the exchange pattern, the producer waits for the reply (*Out* or *Fault* message) to arrive from the endpoint. This step can cause the `process()` method to block indefinitely. However, if the exchange pattern does not mandate a reply, the `process()` method can return immediately after sending the request.

4. When the `process()` method returns, the exchange object contains the reply from the synchronous call (either an *Out* message or a *Fault* message).

Asynchronous producer

Figure 4.7 on page 52 shows an outline of an asynchronous producer, where the producer processes the exchange in a sub-thread, and the preceding processor is not blocked for any significant length of time.

Figure 4.7. Asynchronous Producer



The synchronous producer processes an exchange as follows:

1. Before the processor can call the asynchronous `process()` method, it must create an *asynchronous callback* object, which is responsible for processing the exchange on the return portion of the route. For the asynchronous callback, the processor must implement a class that inherits from the `AsyncCallback` interface.

2. The processor calls the asynchronous `process()` method on the producer to initiate asynchronous processing. The asynchronous `process()` method takes two arguments:
   - an exchange object
   - a synchronous callback object
3. In the body of the `process()` method, the producer creates a `Runnable` object that encapsulates the processing code. The producer then delegates the execution of this `Runnable` object to a sub-thread.
4. The asynchronous `process()` method returns, thereby freeing up the processor's thread. The exchange processing continues in a separate sub-thread.
5. The `Runnable` object sends the *In* message to the endpoint.
6. If required by the exchange pattern, the `Runnable` object waits for the reply (*Out* or *Fault* message) to arrive from the endpoint. The `Runnable` object remains blocked until the reply is received.
7. After the reply arrives, the `Runnable` object inserts the reply (*Out* or *Fault* message) into the exchange object and then calls `done()` on the asynchronous callback object. The asynchronous callback is then responsible for processing the reply message (executed in the sub-thread).

How to Implement a Component

Overview

This section gives a brief overview of the steps required to implement a custom FUSE Mediation Router component.

Which interfaces do you need to implement?

When implementing a component, it is usually necessary to implement the following Java interfaces:

- `org.apache.camel.Component`
- `org.apache.camel.Endpoint`
- `org.apache.camel.Consumer`
- `org.apache.camel.Producer`

In addition, it can also be necessary to implement the following Java interfaces:

- `org.apache.camel.Exchange`
- `org.apache.camel.Message`

Implementation steps

You typically implement a custom component as follows:

1. *Implement the `Component` interface*—A component object acts as an endpoint factory. You extend the `DefaultComponent` class and implement the `createEndpoint()` method.

   See "Component Interface" on page 63.

2. *Implement the `Endpoint` interface*—An endpoint represents a resource identified by a specific URI. The approach taken when implementing an endpoint depends on whether the consumers follow an *event-driven* pattern, a *scheduled poll* pattern, or a *polling* pattern.

   For an event-driven pattern, implement the endpoint by extending the `DefaultEndpoint` class and implementing the following methods:

- `createProducer()`
- `createConsumer()`

For a scheduled poll pattern, implement the endpoint by extending the `ScheduledPollEndpoint` class and implementing the following methods:

- `createProducer()`
- `createConsumer()`

For a polling pattern, implement the endpoint by extending the `DefaultPollingEndpoint` class and implementing the following methods:

- `createProducer()`
- `createPollConsumer()`

See "Endpoint Interface" on page 71.

3. *Implement the `Consumer` interface*—There are several different approaches you can take to implementing a consumer, depending on which pattern you need to implement (event-driven, scheduled poll, or polling). The consumer implementation is also crucially important for determining the threading model used for processing a message exchange.

   See "Implementing the Consumer Interface" on page 92.

4. *Implement the `Producer` interface*—To implement a producer, you extend the `DefaultProducer` class and implement the `process()` method.

   See "Producer Interface" on page 99.

5. *Optionally implement the Exchange or the Message interface*—The default implementations of `Exchange` and `Message` can be used directly, but occasionally, you might find it necessary to customize these types.

See "Exchange Interface" on page 107 and "Message Interface" on page 117.

Installing and configuring the component

You can install a custom component in one of the following ways:

- *Add the component directly to the CamelContext*—The `CamelContext.addComponent()` method adds a component programatically. For more details, see "Adding Components to the Camel Context" in *Deployment Guide*.
- *Add the component using Spring configuration*—The standard Spring `bean` element creates a component instance. The bean's `id` attribute implicitly defines the component prefix. For details, see "Configuring a Component" on page 60.
- *Configure FUSE Mediation Router to auto-discover the component*—Auto-discovery, ensures that FUSE Mediation Router automatically loads the component on demand. For details, see "Setting Up Auto-Discovery" on page 58.

Auto-Discovery and Configuration

Setting Up Auto-Discovery

Overview

Auto-discovery is a mechanism that enables you to dynamically add components to your FUSE Mediation Router application. The component URI prefix is used as a key to load components on demand. For example, if FUSE Mediation Router encounters the endpoint URI, `activemq://MyQName`, and the ActiveMQ endpoint is not yet loaded, FUSE Mediation Router searches for the component identified by the `activemq` prefix and dynamically loads the component.

Availability of component classes

Before configuring auto-discovery, you must ensure that your custom component classes are accessible from your current classpath. Typically, you bundle the custom component classes into a JAR file, and add the JAR file to your classpath.

Configuring auto-discovery

To enable auto-discovery of your component, create a Java properties file named after the component prefix, *`component-prefix`*, and store that file in the following location:

```
/META-INF/services/org/apache/camel/component/component-prefix
```

The *`component-prefix`* properties file must contain the following property setting:

```
class=component-class-name
```

Where *`component-class-name`* is the fully-qualified name of your custom component class. You can also define additional system property settings in this file.

Example

For example, you can enable auto-discovery for the FUSE Mediation Router FTP component by creating the following Java properties file:

```
/META-INF/services/org/apache/camel/component/ftp
```

Which contains the following Java property setting:

```
class=org.apache.camel.component.file.remote.RemoteFileComponent
```

Note

The Java properties file for the FTP component is already defined in the JAR file, `camel-ftp-`*`Version`*`.jar`.

Configuring a Component

Overview

You can add a component by configuring it in the FUSE Mediation Router Spring configuration file, `META-INF/spring/camel-context.xml`. To find the component, the component's URI prefix is matched against the ID attribute of a `bean` element in the Spring configuration. If the component prefix matches a bean element ID, FUSE Mediation Router instantiates the referenced class and injects the properties specified in the Spring configuration.

Note

This mechanism has priority over auto-discovery. If the CamelContext finds a Spring bean with the requisite ID, it will not attempt to find the component using auto-discovery.

Define bean properties on your component class

If there are any properties that you want to inject into your component class, define them as bean properties. For example:

```
public class CustomComponent extends
  DefaultComponent<CustomExchange> {
    ...
    PropType getProperty() { ... }
    void setProperty(PropType v) { ...  }
}
```

The `get`*`Property`*`()` method and the `set`*`Property`*`()` method access the value of *`property`*.

Configure the component in Spring

To configure a component in Spring, edit the configuration file, `META-INF/spring/camel-context.xml`, as shown in Example 4.1 on page 60.

Example 4.1. Configuring a Component in Spring

```
<?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 http://www.springframe
work.org/schema/beans/spring-beans-2.0.xsd
       http://activemq.apache.org/camel/schema/spring http://act
ivemq.apache.org/camel/schema/spring/camel-spring.xsd">
```

```
  <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
    <package>RouteBuilderPackage</package>
  </camelContext>

  <bean id="component-prefix" class="component-class-name">
    <property name="property" value="propertyValue"/>
   </bean>
</beans>
```

The `bean` element with ID *`component-prefix`* configures the *`component-class-name`* component. You can inject properties into the component instance using `property` elements. For example, the `property` element in the preceding example would inject the value, *`propertyValue`*, into the *`property`* property by calling `set`*`Property`*`()` on the component.

Examples

Example 4.2 on page 61 shows an example of how to configure the FUSE Mediation Router's JMS component by defining a bean element with ID equal to `jms`. These settings are added to the Spring configuration file, `camel-context.xml`.

Example 4.2. JMS Component Spring Configuration

```
<?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 http://www.springframe
work.org/schema/beans/spring-beans-2.0.xsd
       http://activemq.apache.org/camel/schema/spring http://act
ivemq.apache.org/camel/schema/spring/camel-spring.xsd">

  <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
    <package>org.apache.camel.example.spring</package> ❶
  </camelContext>

  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent"> ❷
    <property name="connectionFactory"> ❸
       <bean class="org.apache.activemq.ActiveMQConnectionFactory">
         <property name="brokerURL"
                   value="vm://localhost?broker.persistent=false&amp;broker.useJmx=false"/>
 ❹
       </bean>
    </property>
  </bean>
</beans>
```

❶ The `CamelContext` automatically instantiates any `RouteBuilder` classes that it finds in the specified Java package, `org.apache.camel.example.spring`.

❷ The bean element with ID, `jms`, configures the JMS component. The bean ID corresponds to the component's URI prefix. For example, if a route specifies an endpoint with the URI, `jms://MyQName`, FUSE Mediation Router automatically loads the JMS component using the settings from the `jms` bean element.

❸ JMS is just a wrapper for a messaging service. You must specify the concrete implementation of the messaging system by setting the `connectionFactory` property on the `JmsComponent` class.

❹ In this example, the concrete implementation of the JMS messaging service is Apache ActiveMQ. The `brokerURL` property initializes a connection to an ActiveMQ broker instance, where the message broker is embedded in the local Java virtual machine (JVM). If a broker is not already present in the JVM, ActiveMQ will instantiate it with the options `broker.persistent=false` (the broker does not persist messages) and `broker.useJmx=false` (the broker does not open a JMX port).

Chapter 5. Component Interface

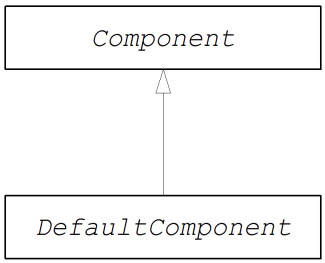
This chapter describes how to implement the `Component` *interface.*

The Component Interface

Overview

To implement a FUSE Mediation Router component, you must implement the `org.apache.camel.Component` interface. An instance of `Component` type provides the entry point into a custom component. That is, all of the other objects in a component are ultimately accessible through the `Component` instance. Figure 5.1 on page 64 shows the relevant Java interfaces and classes that make up the `Component` inheritance hierarchy.

Figure 5.1. Component Inheritance Hierarchy



The Component interface

Example 5.1 on page 64 shows the definition of the `org.apache.camel.Component` interface.

Example 5.1. Component Interface

```
package org.apache.camel;

public interface Component<E extends Exchange> {
    CamelContext getCamelContext();
    void setCamelContext(CamelContext context);

    Endpoint<E> createEndpoint(String uri) throws Exception;
}
```

Component methods

The `Component` interface defines the following methods:

- `getCamelContext()` and `setCamelContext()`—References the `CamelContext` to which this `Component` belongs. The `setCamelContext()`

method is automatically called when you add the component to a `CamelContext`.

- `createEndpoint()`—The factory method that gets called to create `Endpoint` instances for this component. The `uri` parameter is the endpoint URI, which contains the details required to create the endpoint.

Implementing the Component Interface

The DefaultComponent class

You implement a new component by extending the `org.apache.camel.impl.DefaultComponent` class, which provides some standard functionality and default implementations for some of the methods. In particular, the `DefaultComponent` class provides support for URI parsing and for creating a *scheduled executor* (which is used for the scheduled poll pattern).

URI parsing

The `createEndpoint(String uri)` method defined in the base `Component` interface takes a complete, unparsed endpoint URI as its sole argument. The `DefaultComponent` class, on the other hand, defines a three-argument version of the `createEndpoint()` method with the following signature:

```
protected abstract Endpoint<E> createEndpoint(String uri,
                                              String remaining,
                                              Map parameters)
    throws Exception;
```

`uri` is the original, unparsed URI; `remaining` is the part of the URI that remains after stripping off the component prefix at the start and cutting off the query options at the end; and `parameters` contains the parsed query options. It is this version of the `createEndpoint()` method that you must override when inheriting from `DefaultComponent`. This has the advantage that the endpoint URI is already parsed for you.

The following sample endpoint URI for the `file` component shows how URI parsing works in practice:

```
file:///tmp/messages/foo?delete=true&moveNamePostfix=.old
```

For this URI, the following arguments are passed to the three-argument version of `createEndpoint()`:

Header 1	Header 2
`uri`	`file:///tmp/messages/foo?delete=true&moveNamePostfix=.old`
`remaining`	`/tmp/messages/foo`
`parameters`	Two entries are set in `java.util.Map`: • parameter `delete` is boolean `true`

Header 1	Header 2
	• parameter `moveNamePostfix` has the string value, `.old`.

Parameter injection

By default, the parameters extracted from the URI query options are injected on the endpoint's bean properties. The `DefaultComponent` class automatically injects the parameters for you.

For example, if you want to define a custom endpoint that supports two URI query options: `delete` and `moveNamePostfix`. All you must do is define the corresponding bean methods (getters and setters) in the endpoint class:

```
public class FileEndpoint extends ScheduledPollEndpoint<FileEx
change> {
    ...
    public boolean isDelete() {
        return delete;
    }
    public void setDelete(boolean delete) {
        this.delete = delete;
    }
    ...
    public String getMoveNamePostfix() {
        return moveNamePostfix;
    }
    public void setMoveNamePostfix(String moveNamePostfix) {
        this.moveNamePostfix = moveNamePostfix;
    }
}
```

It is also possible to inject URI query options into *consumer* parameters. For details, see "Consumer parameter injection" on page 87.

Disabling endpoint parameter injection

If there are no parameters defined on your `Endpoint` class, you can optimize the process of endpoint creation by disabling endpoint parameter injection. To disable parameter injection on endpoints, override the `useIntrospectionOnEndpoint()` method and implement it to return `false`, as follows:

```
protected boolean useIntrospectionOnEndpoint() {
  return false;
}
```

Note

The `useIntrospectionOnEndpoint()` method does *not* affect the parameter injection that might be performed on a `Consumer` class. Parameter injection at that level is controlled by the `Endpoint.configureProperties()` method (see "Implementing the Endpoint Interface" on page 76).

Scheduled executor service

The `DefaultComponent` class is capable of initializing a *scheduled executor service*, which schedules commands to execute periodically. In particular, the scheduled executor is used in the scheduled poll pattern, where it is responsible for driving the periodic polling of a consumer endpoint.

To instantiate a scheduled executor service, call the `DefaultComponent.getExecutorService()` method, which returns a `java.util.concurrent.ScheduledThreadPoolExecutor` instance that implements the `java.util.concurrent.ScheduledExecutorService` interface). The `ScheduledThreadPoolExecutor` instance is initialized with a thread pool containing five threads. This implies that a scheduled poll consumer can process up to five incoming requests in parallel.

Note

Instantiation of the thread pool is lazy, such that no executor service is created until you actually call `getExecutorService()`.

Validating the URI

If you want to validate the URI before creating an endpoint instance, you can override the `validateURI()` method from the `DefaultComponent` class, which has the following signature:

```
protected void validateURI(String uri,
                           String path,
                           Map parameters)
  throws ResolveEndpointFailedException;
```

If the supplied URI does not have the required format, the implementation of `validateURI()` should throw the `org.apache.camel.ResolveEndpointFailedException` exception.

Creating an endpoint

Example 5.2 on page 69 outlines how to implement the `DefaultComponent.createEndpoint()` method, which is responsible for creating endpoint instances on demand.

Example 5.2. Implementation of `createEndpoint()`

```
public class CustomComponent extends DefaultComponent<CustomExchange> { ❶
    ...
    protected Endpoint<CustomExchange> createEndpoint(String uri, String remaining, Map
parameters) throws Exception { ❷
        CustomEndpoint result = new CustomEndpoint(uri, this); ❸
        // ...
        return result;
    }
}
```

❶ The *`CustomComponent`* is the name of your custom component class, which is defined by extending the `DefaultComponent` class. The type argument, *`CustomExchange`*, can be a custom exchange implementation, but you can also just use `Exchange` here.

❷ When extending `DefaultComponent`, you must implement the `createEndpoint()` method with three arguments (see "URI parsing" on page 66).

❸ Create an instance of your custom endpoint type, *`CustomEndpoint`*, by calling its constructor. At a minimum, this constructor takes a copy of the original URI string, `uri`, and a reference to this component instance, `this`.

Example

Example 5.3 on page 69 shows the complete implementation of the `FileComponent` class, which is taken from the FUSE Mediation Router file component implementation.

Example 5.3. FileComponent Implementation

```
package org.apache.camel.component.file;

import org.apache.camel.CamelContext;
```

```
import org.apache.camel.Endpoint;
import org.apache.camel.impl.DefaultComponent;

import java.io.File;
import java.util.Map;

public class FileComponent extends DefaultComponent<FileExchange> {
    public static final String HEADER_FILE_NAME = "org.apache.camel.file.name";

    public FileComponent() { ❶
    }

    public FileComponent(CamelContext context) { ❷
        super(context);
    }

    protected Endpoint<FileExchange> createEndpoint(String uri, String remaining, Map
parameters) throws Exception { ❸
        File file = new File(remaining);
        FileEndpoint result = new FileEndpoint(file, uri, this);
        return result;
    }
}
```

❶ Always define a no-argument constructor for the component class in order to facilitate automatic instantiation of the class.

❷ A constructor that takes the parent `CamelContext` instance as an argument is convenient when creating a component instance by programming.

❸ The implementation of the `FileComponent.createEndpoint()` method follows the pattern described in Example 5.2 on page 69. The implementation creates a `FileEndpoint` object.

Chapter 6. Endpoint Interface

This chapter describes how to implement the `Endpoint` interface, which is an essential step in the implementation of a FUSE Mediation Router component.

The Endpoint Interface

Overview

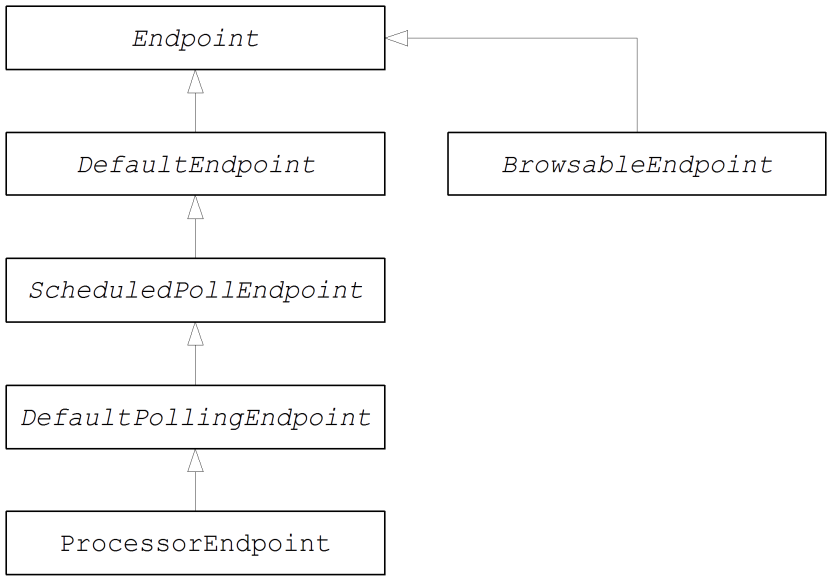
An instance of `org.apache.camel.Endpoint` type encapsulates an endpoint URI, and it also serves as a factory for `Consumer`, `Producer`, and `Exchange` objects. There are three different approaches to implementing an endpoint:

- Event-driven
- scheduled poll
- polling

These endpoint implementation patterns complement the corresponding patterns for implementing a consumer—see "Implementing the Consumer Interface" on page 92.

Figure 6.1 on page 73 shows the relevant Java interfaces and classes that make up the `Endpoint` inheritance hierarchy.

Figure 6.1. Endpoint Inheritance Hierarchy



The Endpoint interface

Example 6.1 on page 73 shows the definition of the `org.apache.camel.Endpoint` interface.

Example 6.1. Endpoint Interface

```
package org.apache.camel;

public interface Endpoint<E extends Exchange> {
    boolean isSingleton();

    String getEndpointUri();

    CamelContext getCamelContext();
    void setCamelContext(CamelContext context);

    void configureProperties(Map options);

    boolean isLenientProperties();

    E createExchange();
    E createExchange(ExchangePattern pattern);
```

```
    E createExchange(Exchange exchange);

    Producer<E> createProducer() throws Exception;

    Consumer<E> createConsumer(Processor processor) throws
Exception;
    PollingConsumer<E> createPollingConsumer() throws Excep
tion;
}
```

Endpoint methods

The `Endpoint` interface defines the following methods:

- `isSingleton()`—Returns `true`, if you want to ensure that each URI maps to a single endpoint within a CamelContext. When this property is `true`, multiple references to the identical URI within your routes always refer to a *single* endpoint instance. When this property is `false`, on the other hand, multiple references to the same URI within your routes refer to *distinct* endpoint instances. Each time you refer to the URI in a route, a new endpoint instance is created.
- `getEndpointUri()`—Returns the endpoint URI of this endpoint.
- `getCamelContext()`—return a reference to the `CamelContext` instance to which this endpoint belongs.
- `setCamelContext()`—Sets the `CamelContext` instance to which this endpoint belongs.
- `configureProperties()`—Stores a copy of the parameter map that is used to inject parameters when creating a new `Consumer` instance.
- `isLenientProperties()`—Returns `true` to indicate that the URI is allowed to contain unknown parameters (that is, parameters that cannot be injected on the `Endpoint` or the `Consumer` class). Normally, this method should be implemented to return `false`.
- `createExchange()`—An overloaded method with the following variants:
  - `E createExchange()`—Creates a new exchange instance with a default exchange pattern setting.
  - `E createExchange(ExchangePattern pattern)`—Creates a new exchange instance with the specified exchange pattern.

- `E createExchange(Exchange exchange)`—Converts the given `exchange` argument to the type of exchange needed for this endpoint.

  If the given exchange is not already of the correct type, this method copies it into a new instance of the correct type. A default implementation of this method is provided in the `DefaultEndpoint` class.

- `createProducer()`—Factory method used to create new `Producer` instances.
- `createConsumer()`—Factory method to create new event-driven consumer instances. The `processor` argument is a reference to the first processor in the route.
- `createPollingConsumer()`—Factory method to create new polling consumer instances.

Endpoint singletons

In order to avoid unnecessary overhead, it is a good idea to create a *single* endpoint instance for all endpoints that have the same URI (within a CamelContext). You can enforce this condition by implementing `isSingleton()` to return `true`.

Note

In this context, *same URI* means that two URIs are the same when compared using string equality. In principle, it is possible to have two URIs that are equivalent, though represented by different strings. In that case, the URIs would not be treated as the same.

Implementing the Endpoint Interface

Alternative ways of implementing an endpoint

The following alternative endpoint implementation patterns are supported:

- "Event-driven endpoint implementation"
- "Scheduled poll endpoint implementation"
- "Polling endpoint implementation"

Event-driven endpoint implementation

If your custom endpoint conforms to the event-driven pattern (see "Consumer Patterns and Threading" on page 47), it is implemented by extending the abstract class, `org.apache.camel.impl.DefaultEndpoint`, as shown in Example 6.2 on page 76.

Example 6.2. Implementing DefaultEndpoint

```
import java.util.Map;
import java.util.concurrent.BlockingQueue;

import org.apache.camel.Component;
import org.apache.camel.Consumer;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultEndpoint;

public class CustomEndpoint extends DefaultEndpoint<CustomExchange> { ❶

    public CustomEndpoint(String endpointUri, Component component) { ❷
        super(endpointUri, component);
        // Do any other initialization...
    }

    public Producer createProducer() throws Exception { ❸
        return new CustomProducer(this);
    }

    public Consumer createConsumer(Processor processor) throws Exception { ❹
        return new CustomConsumer(this, processor);
    }

    public boolean isSingleton() {
        return true;
```

```
    }

    // Implement the following two methods, only if you need a custom exchange class.
    //
    public CustomExchange createExchange() { ❺
        return new CustomExchange(getCamelContext(), getExchangePattern());
    }

    public CustomExchange createExchange(ExchangePattern pattern) {
        return new CustomExchange(getCamelContext(), pattern);
    }
}
```

❶ Implement an event-driven custom endpoint, *`CustomEndpoint`*, by extending the `DefaultEndpoint` class.

❷ You must have at least one constructor that takes the endpoint URI, `endpointUri`, and the parent component reference, `component`, as arguments.

❸ Implement the `createProducer()` factory method to create producer endpoints.

❹ Implement the `createConsumer()` factory method to create event-driven consumer instances.

Important

Do *not* override the `createPollingConsumer()` method.

❺ If you intend to customize the exchange implementation, you should override the `createExchange()` and the `createExchange(ExchangePattern)` methods, to ensure that the correct exchange type is created. If you do not override these methods, the implementations inherited from `DefaultEndpoint` will create a `DefaultExchange` instance.

The `DefaultEndpoint` class provides default implementations of the following methods, which you might find useful when writing your custom endpoint code:

- `getEndpointUri()`—Returns the endpoint URI.
- `getCamelContext()`—Returns a reference to the `CamelContext`.
- `getComponent()`—Returns a reference to the parent component.

- `getExecutorService()`—Returns a reference to a scheduled executor service. The scheduled executor is a `java.util.concurrent.ScheduledExecutorService` object.
- `createPollingConsumer()`—Creates a polling consumer. The created polling consumer's functionality is based on the event-driven consumer. If you override the event-driven consumer method, `createConsumer()`, you get a polling consumer implementation for free.
- `createExchange(Exchange e)`—Converts the given exchange object, `e`, to the type required for this endpoint. This method creates a new endpoint using the overridden `createExchange()` endpoints. This ensures that the method also works for custom exchange types.

Scheduled poll endpoint implementation

If your custom endpoint conforms to the scheduled poll pattern (see "Consumer Patterns and Threading" on page 47) it is implemented by inheriting from the abstract class, `org.apache.camel.impl.ScheduledPollEndpoint`, as shown in Example 6.3 on page 78.

Example 6.3. ScheduledPollEndpoint Implementation

```
import org.apache.camel.Consumer;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.ExchangePattern;
import org.apache.camel.Message;
import org.apache.camel.impl.ScheduledPollEndpoint;

public class CustomEndpoint extends ScheduledPollEndpoint<CustomExchange> { ❶

    protected CustomEndpoint(String endpointUri, CustomComponent component) { ❷
        super(endpointUri, component);
        // Do any other initialization...
    }

    public Producer<CustomExchange> createProducer() throws Exception { ❸
        Producer<CustomExchange> result = new CustomProducer(this);
        return result;
    }

    public Consumer<CustomExchange> createConsumer(Processor processor) throws Exception {
❹
        Consumer<CustomExchange> result = new CustomConsumer(this, processor);
        configureConsumer(result); ❺
```

```
        return result;
    }

    public boolean isSingleton() {
        return true;
    }

    // Implement the following two methods, only if you need a custom exchange class.
    //
    public CustomExchange createExchange() { ❻
        return new CustomExchange(...);
    }

    public CustomExchange createExchange(ExchangePattern pattern) {
        return new CustomExchange(getCamelContext(), pattern);
    }
}
```

❶ Implement a scheduled poll custom endpoint, *`CustomEndpoint`*, by extending the `ScheduledPollEndpoint` class.

❷ You must to have at least one constructor that takes the endpoint URI, `endpointUri`, and the parent component reference, `component`, as arguments.

❸ Implement the `createProducer()` factory method to create a producer endpoint.

❹ Implement the `createConsumer()` factory method to create a scheduled poll consumer instance.

Important

Do *not* override the `createPollingConsumer()` method.

❺ The `configureConsumer()` method, defined in the `ScheduledPollEndpoint` base class, is responsible for injecting consumer query options into the consumer. See "Consumer parameter injection" on page 87.

❻ If you intend to customize the exchange implementation, you should override the `createExchange()` and the `createExchange(ExchangePattern)` methods, to ensure that the correct exchange type is created. If you do not override these methods,

the implementations inherited from `DefaultEndpoint` will create a `DefaultExchange` instance.

Polling endpoint implementation

If your custom endpoint conforms to the polling consumer pattern (see "Consumer Patterns and Threading" on page 47), it is implemented by inheriting from the abstract class, `org.apache.camel.impl.DefaultPollingEndpoint`, as shown in Example 6.4 on page 80.

Example 6.4. DefaultPollingEndpoint Implementation

```
import org.apache.camel.Consumer;
import org.apache.camel.Processor;
import org.apache.camel.Producer;
import org.apache.camel.ExchangePattern;
import org.apache.camel.Message;
import org.apache.camel.impl.DefaultPollingEndpoint;

public class CustomEndpoint extends DefaultPollingEndpoint<CustomExchange> {
    ...
    public PollingConsumer<CustomExchange> createPollingConsumer() throws Exception {
        PollingConsumer<CustomExchange> result = new CustomConsumer(this);
        configureConsumer(result);
        return result;
    }

   // Do NOT implement createConsumer(). It is already implemented in DefaultPollingEndpoint.

    ...
}
```

Because this *`CustomEndpoint`* class is a polling endpoint, you must implement the `createPollingConsumer()` method instead of the `createConsumer()` method. The consumer instance returned from `createPollingConsumer()` must inherit from the `PollingConsumer` interface. For details of how to implement a polling consumer, see "Polling consumer implementation" on page 95.

Apart from the implementation of the `createPollingConsumer()` method, the steps for implementing a `DefaultPollingEndpoint` are similar to the

steps for implementing a `ScheduledPollEndpoint`. See Example 6.3 on page 78 for details.

Implementing the BrowsableEndpoint interface

If you want to expose the list of exchange instances that are pending in the current endpoint, you can implement the `org.apache.camel.spi.BrowsableEndpoint` interface, as shown in Example 6.5 on page 81. It makes sense to implement this interface if the endpoint performs some sort of buffering of incoming events. For example, the FUSE Mediation Router SEDA endpoint implements the `BrowsableEndpoint` interface—see Example 6.6 on page 81.

Example 6.5. BrowsableEndpoint Interface

```
package org.apache.camel.spi;

import java.util.List;

import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;

public interface BrowsableEndpoint<T extends Exchange> extends Endpoint<T> {
    List<Exchange> getExchanges();
}
```

Example

Example 6.6 on page 81 shows the implementation of `SedaEndpoint`, which is taken from the FUSE Mediation Router SEDA component implementation. The SEDA endpoint is an example of an *event-driven endpoint*. Incoming events are stored in a FIFO queue (an instance of `java.util.concurrent.BlockingQueue`) and a SEDA consumer starts up a thread to read and process the events. The events themselves are represented by `org.apache.camel.Exchange` objects.

Example 6.6. SedaEndpoint Implementation

```
package org.apache.camel.component.seda;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.BlockingQueue;

import org.apache.camel.Component;
import org.apache.camel.Consumer;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
```

```
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultEndpoint;
import org.apache.camel.spi.BrowsableEndpoint;

public class SedaEndpoint extends DefaultEndpoint<Exchange> implements BrowsableEndpoint<Ex
change> { ❶
    private BlockingQueue<Exchange> queue;

    public SedaEndpoint(String endpointUri, Component component, BlockingQueue<Exchange>
queue) { ❷
        super(endpointUri, component);
        this.queue = queue;
    }

    public SedaEndpoint(String uri, SedaComponent component, Map parameters) { ❸
        this(uri, component, component.createQueue(uri, parameters));
    }

    public Producer createProducer() throws Exception { ❹
        return new CollectionProducer(this, getQueue());
    }

    public Consumer createConsumer(Processor processor) throws Exception { ❺
        return new SedaConsumer(this, processor);
    }

    public BlockingQueue<Exchange> getQueue() { ❻
        return queue;
    }

    public boolean isSingleton() { ❼
        return true;
    }

    public List<Exchange> getExchanges() { ❽
        return new ArrayList<Exchange>(getQueue());
    }
}
```

❶ The `SedaEndpoint` class follows the pattern for implementing an event-driven endpoint by extending the `DefaultEndpoint` class. The `SedaEndpoint` class also implements the `BrowsableEndpoint` interface, which provides access to the list of exchange objects in the queue.

❷ Following the usual pattern for an event-driven consumer, `SedaEndpoint` defines a constructor that takes an endpoint argument, `endpointUri`, and a component reference argument, `component`.

❸ Another constructor is provided, which delegates queue creation to the parent component instance.

❹ The `createProducer()` factory method creates an instance of `CollectionProducer`, which is a producer implementation that adds events to the queue.

❺ The `createConsumer()` factory method creates an instance of `SedaConsumer`, which is responsible for pulling events off the queue and processing them.

❻ The `getQueue()` method returns a reference to the queue.

❼ The `isSingleton()` method returns `true`, indicating that a single endpoint instance should be created for each unique URI string.

❽ The `getExchanges()` method implements the corresponding abstract method from `BrowsableEndpoint`.

Chapter 7. Consumer Interface

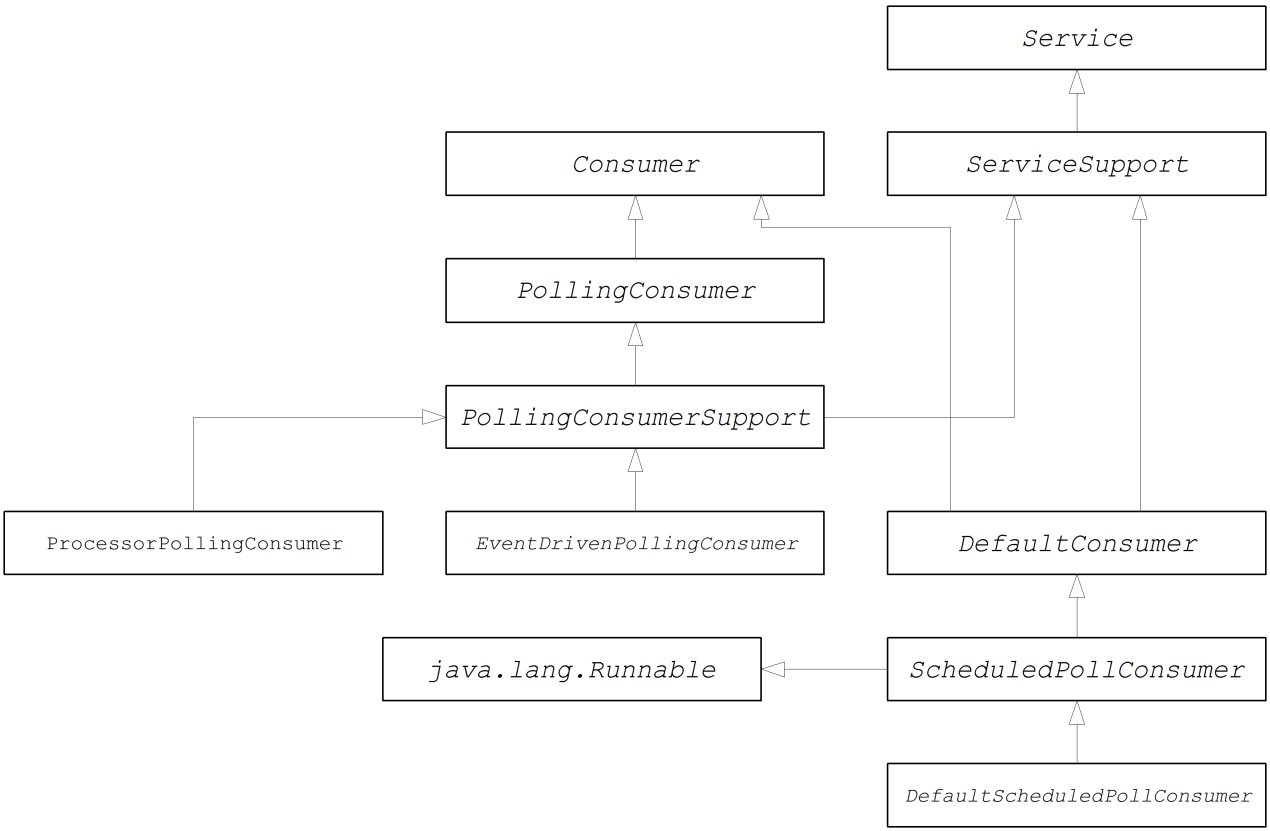
This chapter describes how to implement the `Consumer` interface, which is an essential step in the implementation of a FUSE Mediation Router component.

The Consumer Interface

Overview

An instance of `org.apache.camel.Consumer` type represents a source endpoint in a route. There are several different ways of implementing a consumer (see "Consumer Patterns and Threading" on page 47), and this degree of flexibility is reflected in the inheritance hierarchy (see Figure 7.1 on page 87), which includes several different base classes for implementing a consumer.

Figure 7.1. Consumer Inheritance Hierarchy



Consumer parameter injection

For consumers that follow the scheduled poll pattern (see "Scheduled poll pattern" on page 48), FUSE Mediation Router provides support for injecting parameters into consumer instances. For example, consider the following endpoint URI for a component identified by the `custom` prefix:

```
custom:destination?consumer.myConsumerParam
```

FUSE Mediation Router provides support for automatically injecting query options of the form `consumer.*`. For the `consumer.myConsumerParam` parameter, you need to define corresponding setter and getter methods on the `Consumer` implementation class as follows:

```
public class CustomConsumer<E extends Exchange> extends ScheduledPollConsumer<E> {
    ...
    String getMyConsumerParam() { ... }
    void setMyConsumerParam(String s) { ... }
    ...
}
```

Where the getter and setter methods follow the usual Java bean conventions (including capitalizing the first letter of the property name).

In addition to defining the bean methods in your Consumer implementation, you must also remember to call the `configureConsumer()` method in the implementation of `Endpoint.createConsumer()`. See "Scheduled poll endpoint implementation" on page 78). Example 7.1 on page 88 shows an example of a `createConsumer()` method implementation, taken from the `FileEndpoint` class in the file component:

Example 7.1. FileEndpoint createConsumer() Implementation

```
...
public class FileEndpoint extends ScheduledPollEndpoint<FileExchange> {
    ...
    public Consumer<FileExchange> createConsumer(Processor processor) throws Exception {
        Consumer<FileExchange> result = new FileConsumer(this, processor);
        configureConsumer(result);
        return result;
    }
    ...
    }
```

At run time, consumer parameter injection works as follows:

1. When the endpoint is created, the default implementation of `DefaultComponent.createEndpoint(String uri)` parses the URI to extract the consumer parameters, and stores them in the endpoint instance by calling `ScheduledPollEndpoint.configureProperties()`.

2. When `createConsumer()` is called, the method implementation calls `configureConsumer()` to inject the consumer parameters (see Example 7.1 on page 88).

3. The `configureConsumer()` method uses Java reflection to call the setter methods whose names match the relevant options after the `consumer.` prefix has been stripped off.

Scheduled poll parameters

A consumer that follows the scheduled poll pattern automatically supports the consumer parameters shown in Table 7.1 on page 89 (which can appear as query options in the endpoint URI).

Table 7.1. Scheduled Poll Parameters

Name	Default	Description
`initialDelay`	`1000`	Delay, in milliseconds, before the first poll.
`delay`	`500`	Depends on the value of the `useFixedDelay` flag (time unit is milliseconds).
`useFixedDelay`	`false`	If `false`, the `delay` parameter is interpreted as the polling period. Polls will occur at `initialDelay`, `initialDelay+delay`, `initialDelay+2*delay`, and so on. If `true`, the `delay` parameter is interpreted as the time elapsed between the previous execution and the next execution. Polls will occur at `initialDelay`, `initialDelay+[`*`ProcessingTime`*`]+delay`, and so on. Where *`ProcessingTime`* is the time taken to process an exchange object in the current thread.

Converting between event-driven and polling consumers

FUSE Mediation Router provides two special consumer implementations which can be used to convert back and forth between an event-driven consumer and a polling consumer. The following conversion classes are provided:

- `org.apache.camel.impl.EventDrivenPollingConsumer`—Converts an event-driven consumer into a polling consumer instance.
- `org.apache.camel.impl.DefaultScheduledPollConsumer`—Converts a polling consumer into an event-driven consumer instance.

In practice, these classes are used to simplify the task of implementing an `Endpoint` type. The `Endpoint` interface defines the following two methods for creating a consumer instance:

```
package org.apache.camel;

public interface Endpoint<E extends Exchange> {
    ...
    Consumer<E> createConsumer(Processor processor) throws
Exception;
    PollingConsumer<E> createPollingConsumer() throws Excep
tion;
}
```

`createConsumer()` returns an event-driven consumer and `createPollingConsumer()` returns a polling consumer. You would only implement one these methods. For example, if you are following the event-driven pattern for your consumer, you would implement the `createConsumer()` method provide a method implementation for `createPollingConsumer()` that simply raises an exception. With the help of the conversion classes, however, FUSE Mediation Router is able to provide a more useful default implementation.

For example, if you want to implement your consumer according to the event-driven pattern, you implement the endpoint by extending `DefaultEndpoint` and implementing the `createConsumer()` method. The implementation of `createPollingConsumer()` is inherited from `DefaultEndpoint`, where it is defined as follows:

```
public PollingConsumer<E> createPollingConsumer() throws Ex
ception {
    return new EventDrivenPollingConsumer<E>(this);
}
```

The `EventDrivenPollingConsumer` constructor takes a reference to the event-driven consumer, `this`, effectively wrapping it and converting it into a polling consumer. To implement the conversion, the `EventDrivenPollingConsumer` instance buffers incoming events and makes them available on demand through the `receive()`, the `receive(long timeout)`, and the `receiveNoWait()` methods.

Analogously, if you are implementing your consumer according to the polling pattern, you implement the endpoint by extending `DefaultPollingEndpoint` and implementing the `createPollingConsumer()` method. In this case, the implementation of the `createConsumer()` method is inherited from `DefaultPollingEndpoint`, and the default implementation returns a

`DefaultScheduledPollConsumer` instance (which converts the polling consumer into an event-driven consumer).

Implementing the Consumer Interface

Alternative ways of implementing a consumer

You can implement a consumer in one of the following ways:

- "Event-driven consumer implementation"
- "Scheduled poll consumer implementation"
- "Polling consumer implementation"

Event-driven consumer implementation

In an event-driven consumer, processing is driven explicitly by external events. The events are received through an event-listener interface, where the listener interface is specific to the particular event source.

Example 7.2 on page 92 shows the implementation of the `JMXConsumer` class, which is taken from the FUSE Mediation Router JMX component implementation. The `JMXConsumer` class is an example of an event-driven consumer, which is implemented by inheriting from the `org.apache.camel.impl.DefaultConsumer` class. In the case of the `JMXConsumer` example, events are represented by calls on the `NotificationListener.handleNotification()` method, which is a standard way of receiving JMX events. In order to receive these JMX events, it is necessary to implement the `NotificationListener` interface and override the `handleNotification()` method, as shown in Example 7.2 on page 92.

Example 7.2. JMXConsumer Implementation

```
package org.apache.camel.component.jmx;

import javax.management.Notification;
import javax.management.NotificationListener;
import org.apache.camel.Processor;
import org.apache.camel.impl.DefaultConsumer;

public class JMXConsumer extends DefaultConsumer implements
NotificationListener { ❶

    JMXEndpoint jmxEndpoint;

    public JMXConsumer(JMXEndpoint endpoint, Processor pro
cessor) { ❷
        super(endpoint, processor);
```

```
        this.jmxEndpoint = endpoint;
    }

   public void handleNotification(Notification notification,
 Object handback) { ❸
        try {
            getProcessor().process(jmxEndpoint.createEx
change(notification)); ❹
        } catch (Throwable e) {
            handleException(e); ❺
        }
    }
}
```

❶ The `JMXConsumer` pattern follows the usual pattern for event-driven consumers by extending the `DefaultConsumer` class. Additionally, because this consumer is designed to receive events from JMX (which are represented by JMX notifications), it is necessary to implement the `NotificationListener` interface.

❷ You must implement at least one constructor that takes a reference to the parent endpoint, `endpoint`, and a reference to the next processor in the chain, `processor`, as arguments.

❸ The `handleNotification()` method (which is defined in `NotificationListener`) is automatically invoked by JMX whenever a JMX notification arrives. The body of this method should contain the code that performs the consumer's event processing. Because the `handleNotification()` call originates from the JMX layer, the consumer's threading model is implicitly controlled by the JMX layer, not by the `JMXConsumer` class.

Note

The `handleNotification()` method is specific to the JMX example. When implementing your own event-driven consumer, you must identify an analogous event listener method to implement in your custom consumer.

❹ This line of code combines two steps. First, the JMX notification object is converted into an exchange object, which is the generic representation of an event in FUSE Mediation Router. The, the newly created exchange object is passed to the next processor in the route (invoked synchronously).

❺ The `handleException()` method is implemented by the `DefaultConsumer` base class. By default, it handles exceptions using the `org.apache.camel.impl.LoggingExceptionHandler` class.

Scheduled poll consumer implementation

In a scheduled poll consumer, polling events are automatically generated by a timer class, `java.util.concurrent.ScheduledExecutorService`. To receive the generated polling events, you must implement the `ScheduledPollConsumer.poll()` method (see "Consumer Patterns and Threading" on page 47).

Example 7.3 on page 94 shows how to implement a consumer that follows the scheduled poll pattern, which is implemented by extending the `ScheduledPollConsumer` class.

Example 7.3. ScheduledPollConsumer Implementation

```
import java.util.concurrent.ScheduledExecutorService;

import org.apache.camel.Consumer;
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Message;
import org.apache.camel.PollingConsumer;
import org.apache.camel.Processor;

import org.apache.camel.impl.ScheduledPollConsumer;

public class CustomConsumer<E extends Exchange> extends ScheduledPollConsumer<E> { ❶
    private final CustomEndpoint endpoint;

    public CustomConsumer(CustomEndpoint endpoint, Processor processor) { ❷
        super(endpoint, processor);
        this.endpoint = endpoint;
    }

    protected void poll() throws Exception { ❸
        E exchange = /* Receive exchange object ... */;

        // Example of a synchronous processor.
        getProcessor().process(exchange); ❹
    }

    @Override
    protected void doStart() throws Exception { ❺
```

```
        // Pre-Start:
        // Place code here to execute just before start of processing.
        super.doStart();
        // Post-Start:
        // Place code here to execute just after start of processing.
    }

    @Override
    protected void doStop() throws Exception { ❻
        // Pre-Stop:
        // Place code here to execute just before processing stops.
        super.doStop();
        // Post-Stop:
        // Place code here to execute just after processing stops.
    }
}
```

❶ Implement a scheduled poll consumer class, *`CustomConsumer`*, by extending the `org.apache.camel.impl.ScheduledPollConsumer` class.

❷ You must implement at least one constructor that takes a reference to the parent endpoint, `endpoint`, and a reference to the next processor in the chain, `processor`, as arguments.

❸ Override the `poll()` method to receive the scheduled polling events.

This is where you should put the code that retrieves and processes incoming events (represented by exchange objects).

❹ In this example, the event is processed synchronously. If you want to process events asynchronously, you should use a reference to an asynchronous processor instead, by calling `getAsyncProcessor()`. For details of how to process events asynchronously, see "Asynchronous Processing" on page 51.

❺ *(Optional)* If you want some lines of code to execute as the consumer is starting up, override the `doStart()` method as shown.

❻ *(Optional)* If you want some lines of code to execute as the consumer is stopping, override the `doStop()` method as shown.

Polling consumer implementation

Example 7.4 on page 96 outlines how to implement a consumer that follows the polling pattern, which is implemented by extending the `PollingConsumerSupport` class.

Example 7.4. PollingConsumerSupport Implementation

```
import org.apache.camel.Exchange;
import org.apache.camel.RuntimeCamelException;
import org.apache.camel.impl.PollingConsumerSupport;

public class CustomConsumer extends PollingConsumerSupport {
❶
    private final CustomEndpoint endpoint;

    public CustomConsumer(CustomEndpoint endpoint) { ❷
        super(endpoint);
        this.endpoint = endpoint;
    }

    public Exchange receiveNoWait() { ❸
       Exchange exchange = /* Obtain an exchange object. */;

        // Further processing ...
        return exchange;
    }

    public Exchange receive() { ❹
        // Blocking poll ...
    }

    public Exchange receive(long timeout) { ❺
        // Poll with timeout ...
    }

    protected void doStart() throws Exception { ❻
        // Code to execute whilst starting up.
    }

    protected void doStop() throws Exception {
        // Code to execute whilst shutting down.
    }
}
```

❶ Implement your polling consumer class, *CustomConsumer*, by extending the `org.apache.camel.impl.PollingConsumerSupport` class.

❷ You must implement at least one constructor that takes a reference to the parent endpoint, `endpoint`, as an argument. A polling consumer does not need a reference to a processor instance.

❸ The `receiveNoWait()` method should implement a non-blocking algorithm for retrieving an event (exchange object). If no event is available, it should return `null`.

❹ The `receive()` method should implement a blocking algorithm for retrieving an event. This method can block indefinitely, if events remain unavailable.

❺ The `receive(long timeout)` method implements an algorithm that can block for as long as the specified timeout (typically specified in units of milliseconds).

❻ If you want to insert code that executes while a consumer is starting up or shutting down, implement the `doStart()` method and the `doStop()` method, respectively.

Chapter 8. Producer Interface

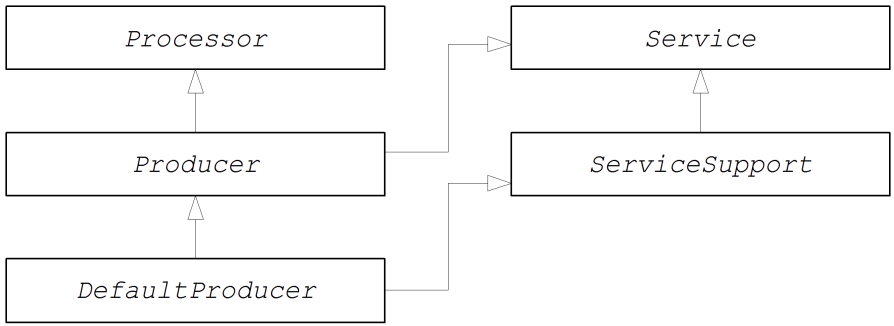
This chapter describes how to implement the `Producer` interface, which is an essential step in the implementation of a FUSE Mediation Router component.

The Producer Interface

Overview

An instance of `org.apache.camel.Producer` type represents a target endpoint in a route. The role of the producer is to send requests (*In* messages) to a specific physical endpoint and to receive the corresponding response (*Out* or *Fault* message). A `Producer` object is essentially a special kind of `Processor` that appears at the end of a processor chain (equivalent to a route). Figure 8.1 on page 100 shows the inheritance hierarchy for producers.

Figure 8.1. Producer Inheritance Hierarchy



The Producer interface

Example 8.1 on page 100 shows the definition of the `org.apache.camel.Producer` interface.

Example 8.1. Producer Interface

```
package org.apache.camel;

public interface Producer<E extends Exchange> extends Pro
cessor, Service {

    Endpoint<E> getEndpoint();

    E createExchange();

    E createExchange(ExchangePattern pattern);
```

```
    E createExchange(E exchange);
}
```

Producer methods

The `Producer` interface defines the following methods:

- `process()` *(inherited from Processor)*—The most important method. A producer is essentially a special type of processor that sends a request to an endpoint, instead of forwarding the exchange object to another processor. By overriding the `process()` method, you define how the producer sends and receives messages to and from the relevant endpoint.
- `getEndpoint()`—Returns a reference to the parent endpoint instance.
- `createExchange()`—These overloaded methods are analogous to the corresponding methods defined in the `Endpoint` interface. Normally, these methods delegate to the corresponding methods defined on the parent `Endpoint` instance (this is what the `DefaultEndpoint` class does by default). Occasionally, you might need to override these methods.

Asynchronous processing

Processing an exchange object in a producer—which usually involves sending a message to a remote destination and waiting for a reply—can potentially block for a significant length of time. If you want to avoid blocking the current thread, you can opt to implement the producer as an *asynchronous processor*. The asynchronous processing pattern decouples the preceding processor from the producer, so that the `process()` method returns without delay. See "Asynchronous Processing" on page 51.

When implementing a producer, you can support the asynchronous processing model by implementing the `org.apache.camel.AsyncProcessor` interface. On its own, this is not enough to ensure that the asynchronous processing model will be used: it is also necessary for the preceding processor in the chain to call the asynchronous version of the `process()` method. The definition of the `AsyncProcessor` interface is shown in Example 8.2 on page 101.

Example 8.2. AsyncProcessor Interface

```
package org.apache.camel;

public interface AsyncProcessor extends Processor {
    boolean process(Exchange exchange, AsyncCallback callback);
}
```

The asynchronous version of the `process()` method takes an extra argument, `callback`, of `org.apache.camel.AsyncCallback` type. The corresponding `AsyncCallback` interface is defined as shown in Example 8.3 on page 102.

Example 8.3. AsyncCallback Interface

```
package org.apache.camel;

public interface AsyncCallback {
    void done(boolean doneSynchronously);
}
```

The caller of `AsyncProcessor.process()` must provide an implementation of `AsyncCallback` to receive the notification that processing has finished. The `AsyncCallback.done()` method takes a boolean argument that indicates whether the processing was performed synchronously or not. Normally, the flag would be `false`, to indicate asynchronous processing. In some cases, however, it can make sense for the producer *not* to process asynchronously (in spite of being asked to do so). For example, if the producer knows that the processing of the exchange will complete rapidly, it could optimise the processing by doing it synchronously. In this case, the `doneSynchronously` flag should be set to `true`.

ExchangeHelper class

When implementing a producer, you might find it helpful to call some of the methods in the `org.apache.camel.util.ExchangeHelper` utility class. For full details of the `ExchangeHelper` class, see "The ExchangeHelper Class" on page 31.

Implementing the Producer Interface

Alternative ways of implementing a producer

You can implement a producer in one of the following ways:

- "How to implement a synchronous producer" .
- "How to implement an asynchronous producer" .

How to implement a synchronous producer

Example 8.4 on page 103 outlines how to implement a synchronous producer. In this case, call to `Producer.process()` blocks until a reply (either an *Out* message or a *Fault* message) is received.

Example 8.4. DefaultProducer Implementation

```
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultProducer;

public class CustomProducer extends DefaultProducer { ❶

    public CustomProducer(Endpoint endpoint) { ❷
        super(endpoint);
        // Perform other initialization tasks...
    }

    public void process(Exchange exchange) throws Exception
{ ❸
        // Process exchange synchronously.
        // ...
    }
}
```

❶ Implement a custom synchronous producer class, *`CustomProducer`*, by extending the `org.apache.camel.impl.DefaultProducer` class.

❷ Implement a constructor that takes a reference to the parent endpoint.

❸ The `process()` method implementation represents the core of the producer code. The implementation of the `process()` method is entirely dependent on the type of component that you are implementing. In outline, the `process()` method is normally implemented as follows:

- If the exchange contains an *In* message, and if this is consistent with the specified exchange pattern, then send the *In* message to the designated endpoint.
- If the exchange pattern anticipates the receipt of an *Out* message or a *Fault* message, then wait until the *Out* message or the *Fault* message has been received. This typically causes the `process()` method to block for a significant length of time.
- When a reply is received, call either `exchange.setOut()` or `exchange.setFault()` to attach the reply to the exchange object and then return.

How to implement an asynchronous producer

Example 8.5 on page 104 outlines how to implement an asynchronous producer. In this case, you must implement both a synchronous `process()` method and an asynchronous `process()` method (which takes an additional `AsyncCallback` argument).

Example 8.5. CollectionProducer Implementation

```
import org.apache.camel.AsyncCallback;
import org.apache.camel.AsyncProcessor;
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Producer;
import org.apache.camel.impl.DefaultProducer;

public class CustomProducer extends DefaultProducer implements AsyncProcessor { ❶

    public CustomProducer(Endpoint endpoint) { ❷
        super(endpoint);
        // ...
    }

    public void process(Exchange exchange) throws Exception { ❸
        // Process exchange synchronously.
        // ...
    }

    public boolean process(Exchange exchange, AsyncCallback callback) { ❹
        // Process exchange asynchronously.
        CustomProducerTask task = new CustomProducerTask(exchange, callback);
        // Process 'task' in a separate thread...
        // ...
```

```
        return false; ❺
    }
}

public class CustomProducerTask implements Runnable { ❻
    private Exchange exchange;
    private AsyncCallback callback;

    public CustomProducerTask(Exchange exchange, AsyncCallback callback) {
        this.exchange = exchange;
        this.callback = callback;
    }

    public void run() { ❼
        // Process exchange.
        // ...
        callback.done(false);
    }
}
```

❶ Implement a custom asynchronous producer class, *CustomProducer*, by extending the `org.apache.camel.impl.DefaultProducer` class, and implementing the `AsyncProcessor` interface.

❷ Implement a constructor that takes a reference to the parent endpoint.

❸ Implement the synchronous `process()` method.

❹ Implement the asynchronous `process()` method. You can implement the asynchronous method in several ways. The approach shown here is to create a `java.lang.Runnable` instance, `task`, that represents the code that runs in a sub-thread. You then use the Java threading API to run the task in a sub-thread (for example, by creating a new thread or by allocating the task to an existing thread pool).

❺ Normally, you return `false` from the asynchronous `process()` method, to indicate that the exchange was processed asynchronously.

❻ The *CustomProducer*`Task` class encapsulates the processing code that runs in a sub-thread. This class must store a copy of the `Exchange` object, `exchange`, and the `AsyncCallback` object, `callback`, as private member variables.

❼ The `run()` method contains the code that sends the *In* message to the producer endpoint and waits to receive the reply, if any. After receiving the reply (*Out* message or *Fault* message) and inserting it into the

exchange object, you must call `callback.done()` to notify the caller that processing is complete.

Chapter 9. Exchange Interface

This chapter describes how to implement the `Exchange` interface, which is an optional step in the implementation of a FUSE Mediation Router component.

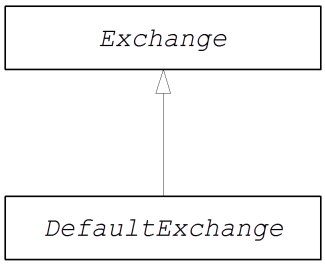
The Exchange Interface

Overview

An instance of `org.apache.camel.Exchange` type encapsulates all of the messages belonging to a single message exchange. For example, a typical synchronous invocation consists of an *In* message and an *Out* message.

Figure 9.1 on page 108 shows the inheritance hierarchy for the exchange type. You do not always need to implement a custom exchange type for a component. In many cases, the default implementation, `DefaultExchange`, is adequate.

Figure 9.1. Exchange Inheritance Hierarchy



The Exchange interface

Example 9.1 on page 108 shows the definition of the `org.apache.camel.Exchange` interface.

Example 9.1. Exchange Interface

```
package org.apache.camel;

import java.util.Map;

import org.apache.camel.spi.UnitOfWork;

public interface Exchange {
    ExchangePattern getPattern();

    Object getProperty(String name);
    <T> T  getProperty(String name, Class<T> type);
    void   setProperty(String name, Object value);
    Object removeProperty(String name);
    Map<String, Object> getProperties();
```

```
    Message getIn();
    void    setIn(Message in);

    Message getOut();
    Message getOut(boolean lazyCreate);
    void setOut(Message out);

    Message getFault();
    Message getFault(boolean lazyCreate);

    Throwable getException();
    void      setException(Throwable e);

    boolean isFailed();

    CamelContext getContext();

    Exchange newInstance();

    Exchange copy();

    void copyFrom(Exchange source);

    UnitOfWork getUnitOfWork();
    void setUnitOfWork(UnitOfWork unitOfWork);

    String getExchangeId();
    void setExchangeId(String id);
}
```

Exchange methods

The `Exchange` interface defines the following methods:

- `getPattern()`—The exchange pattern can be one of the values enumerated in `org.apache.camel.ExchangePattern`. The following exchange pattern values are supported:
  - `InOnly`
  - `RobustInOnly`
  - `InOut`
  - `InOptionalOut`
  - `OutOnly`

- `RobustOutOnly`
- `OutIn`
- `OutOptionalIn`

Normally, you specify the exchange pattern value in the constructor of your custom exchange class.

- `setProperty()`, `getProperty()`, `getProperties()`, `removeProperty()`—Use the property setter and getter methods to associate named properties with the exchange instance. The properties consist of miscellaneous metadata that you might need for your custom exchange implementation.
- `setIn()`, `getIn()`—Setter and getter methods for the *In* message. These methods are used only for exchange patterns that can have an *In* message.

  The `getIn()` implementation provided by the `DefaultExchange` class implements lazy creation semantics: if the *`In`* message is null when `getIn()` is called, the `DefaultExchange` class creates a default *In* message.
- `setOut()`, `getOut()`—Setter and getter methods for the *Out* message. These methods are used only for exchange patterns that can have an *Out* message.

  There are two varieties of `getOut()` method in the `DefaultExchange` class:

  - `getOut()` with no arguments enables lazy creation of an *Out* message. If the current *Out* message is `null`, a new message is automatically created.[1]
  - `getOut(boolean lazyCreate)` with a boolean argument triggers lazy creation. If the argument is `true`, it returns the current value even if the current *Out* message is `null`.
- `getFault()`—Getter message for the fault message. There are two varieties of `getFault()` method in the `DefaultExchange` class:

[1] The `DefaultExchange` class also defines a `setFault()` method.

  - `getFault()` with no arguments enables lazy creation of a *Fault* message.
  - `getFault(boolean lazyCreate)` with a boolean argument triggers lazy creation. If the argument is `true`, it returns the current value even if the current *Fault* message is `null`.

- `setException()`, `getException()`—Getter and setter methods for an exception object (of `Throwable` type).
- `isFailed()`—Returns `true`, if the exchange failed either due to an exception or due to a fault.
- `getContext()`—Returns a reference to the associated `CamelContext` instance.
- `newInstance()`—Creates a new exchange instance for the purpose of copying the current exchange object. For example, in the `DefaultExchange` class, the `copy()` method calls `newInstance()` to create a new exchange instance.
- `copy()`—Creates a new, identical (apart from the exchange ID) copy of the current custom exchange object. The body and headers of the *In* message, the *Out* message (if any), and the *Fault* message (if any) are also copied by this operation.
- `copyFrom()`—Copies the generic contents (apart from the exchange ID) of the specified generic exchange object, `exchange`, into the current exchange instance. Because this method must be able to copy from *any* exchange type, it copies the generic exchange properties, but not the custom properties. The body and headers of the *In* message, the *Out* message (if any), and the *Fault* message (if any) are also copied by this operation.
- `setUnitOfWork()`, `getUnitOfWork()`—Getter and setter methods for the `org.apache.camel.spi.UnitOfWork` bean property. This property is only required for exchanges that can participate in a transaction.
- `setExchangeId()`, `getExchangeId()`—Getter and setter methods for the exchange ID. Whether or not a custom component uses and exchange ID is an implementation detail.

Implementing the Exchange Interface

How to implement a custom exchange

Example 9.2 on page 112 outlines how to implement an exchange by extending the `DefaultExchange` class.

Example 9.2. Custom Exchange Implementation

```
import org.apache.camel.CamelContext;
import org.apache.camel.Exchange;
import org.apache.camel.ExchangePattern;
import org.apache.camel.impl.DefaultExchange;

public class CustomExchange extends DefaultExchange { ❶

    public CustomExchange(CamelContext camelContext, Exchange
Pattern pattern) { ❷
        super(camelContext, pattern);
        // Set other member variables...
    }

    public CustomExchange(CamelContext camelContext) { ❸
        super(camelContext);
        // Set other member variables...
    }

    public CustomExchange(DefaultExchange parent) { ❹
        super(parent);
        // Set other member variables...
    }

    @Override
    public Exchange newInstance() { ❺
        Exchange e = new CustomExchange(this);
        // Copy custom member variables from current in
stance...
        return e;
    }

    @Override
    protected Message createInMessage() { ❻
        return new CustomMessage();
    }

    @Override
    protected Message createOutMessage() {
        return new CustomMessage();
```

```
    }

    @Override
    protected Message createFaultMessage() {
        return new CustomMessage();
    }

    @Override
    protected void configureMessage(Message message) { ❼
        super.configureMessage(message);
        // Perform custom message configuration...
    }
}
```

❶ Implements a custom exchange class, *CustomExchange*, by extending the `org.apache.camel.impl.DefaultExchange` class.

❷ You usually need a constructor that lets you specify the exchange pattern explicitly, as shown here.

❸ This constructor, taking only a `CamelContext` argument, `context`, implicitly sets the exchange pattern to `InOnly` as defined in the `DefaultExchange` constructor.

❹ This constructor copies the exchange pattern and the unit of work from the specified exchange object, `parent`.

❺ The `newInstance()` method is called from inside the `DefaultExchange.copy()` method. Customization of the `newInstance()` method should focus on copying all of the *custom* properties of the current exchange instance into the new exchange instance. The `DefaultExchange.copy()` method copies the generic exchange properties (by calling `copyFrom()`).

❻ *(Optional)* Only needed if you implement a custom message type. The `createInMessage()`, `createOutMessage()`, and `createFaultMessage()` methods are implemented to support lazy message creation when you are using a custom message type. In this example, `createInMessage()` returns a message of *CustomMessage* type. When a new *In* message is created by a call to `getIn()`, the default `getIn()` implementation calls `createInMessage()` to create the new message.

❼ In the body of `configureMessage()` you can put code to configure all message types (*In*, *Out*, and *Fault*). The `DefaultExchange` class uses

`configureMessage()` to configure a message whenever you call `setIn()`, `setOut()`, or `setFault()`, and whenever a message is created by lazy instantiation.

Example

Example 9.3 on page 114 shows the implementation of the `FileExchange` class, which is taken from the FUSE Mediation Router file component implementation. The `FileExchange` implementation is characterised by two things:

- It has an additional `file` property, which references the file containing the *In* message,
- It only supports the `InOnly` exchange pattern.

Example 9.3. FileExchange Implementation

```
package org.apache.camel.component.file;

import org.apache.camel.CamelContext;
import org.apache.camel.Exchange;
import org.apache.camel.ExchangePattern;
import org.apache.camel.impl.DefaultExchange;

import java.io.File;

public class FileExchange extends DefaultExchange {
    private File file;

    public FileExchange(CamelContext camelContext, Exchange
Pattern pattern, File file) { ❶
        super(camelContext, pattern);
        setIn(new FileMessage(file));
        this.file = file;
    }

    public FileExchange(DefaultExchange parent, File file) {
 ❷
        super(parent);
        this.file = file;
    }

    public File getFile() { ❸
        return this.file;
    }
```

```
    public void setFile(File file) {
        this.file = file;
    }

    public Exchange newInstance() { ❹
        return new FileExchange(this, getFile());
    }
}
```

❶ In addition to letting you specify the Camel context, `camelContext`, and the exchange pattern, `pattern`, this constructor also specifies the custom property, `file`.

❷ This constructor gets called by the `newInstance()` method. This constructor copies the unit of work and the exchange pattern from `parent` (implemented by the super-constructor) and initializes the file property with the specified value.

❸ The `getFile()` and `setFile()` methods access the `file` property, which represents the file from which the exchange object reads the *In* message.

❹ The `newInstance()` method is overridden to ensure that the `DefaultExchange.copy()` method works properly. The form of constructor called here ensures that the `file` property gets copied into the new instance.

Chapter 10. Message Interface

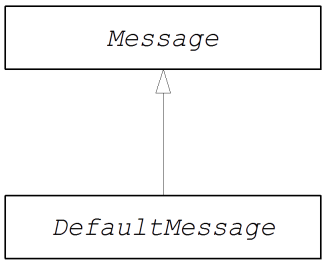
This chapter describes how to implement the `Message` interface, which is an optional step in the implementation of a FUSE Mediation Router component.

The Message Interface

Overview

An instance of `org.apache.camel.Message` type can represent any kind of message (*In*, *Out*, or *Fault*). Figure 10.1 on page 118 shows the inheritance hierarchy for the message type. You do not always need to implement a custom message type for a component. In many cases, the default implementation, `DefaultMessage`, is adequate.

Figure 10.1. Message Inheritance Hierarchy



The Message interface

Example 10.1 on page 118 shows the definition of the `org.apache.camel.Message` interface.

Example 10.1. Message Interface

```
package org.apache.camel;

import java.util.Map;
import java.util.Set;

import javax.activation.DataHandler;

public interface Message {

    String getMessageId();
    void setMessageId(String messageId);

    Exchange getExchange();

    Object getHeader(String name);
    <T> T getHeader(String name, Class<T> type);
    void setHeader(String name, Object value);
    Object removeHeader(String name);
```

```
    Map<String, Object> getHeaders();
    void setHeaders(Map<String, Object> headers);

    Object getBody();
    <T> T  getBody(Class<T> type);
    void     setBody(Object body);
    <T> void setBody(Object body, Class<T> type);

    DataHandler getAttachment(String id);
    Map<String, DataHandler> getAttachments();
    Set<String> getAttachmentNames();
    void removeAttachment(String id);
    void addAttachment(String id, DataHandler content);
   void setAttachments(Map<String, DataHandler> attachments);

    boolean hasAttachments();

    Message copy();

    void copyFrom(Message message);
}
```

Message methods

The `Message` interface defines the following methods:

- `setMessageId()`, `getMessageId()`—Getter and setter methods for the message ID. Whether or not you need to use a message ID in your custom component is an implementation detail.
- `getExchange()`—Returns a reference to the parent exchange object.
- `getHeader()`, `getHeaders()`, `setHeader()`, `setHeaders()`, `removeHeader()`—Getter and setter methods for the message headers.

  In general, these message headers can be used either to store actual header data, or to store miscellaneous metadata.
- `getBody()`, `setBody()`—Getter and setter methods for the message body.
- `getAttachment()`, `getAttachments()`, `getAttachmentNames()`, `removeAttachment()`, `addAttachment()`, `setAttachments()`, `hasAttachments()`—Methods to get, set, add, and remove attachments.

- `copy()`—Creates a new, identical (including the message ID) copy of the current custom message object.
- `copyFrom()`—Copies the complete contents (including the message ID) of the specified generic message object, `message`, into the current message instance. Because this method must be able to copy from *any* message type, it copies the generic message properties, but not the custom properties.

Implementing the Message Interface

How to implement a custom message

Example 10.2 on page 121 outlines how to implement a message by extending the `DefaultMessage` class.

Example 10.2. Custom Message Implementation

```
import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultMessage;

public class CustomMessage extends DefaultMessage { ❶

    public CustomMessage() { ❷
        // Create message with default properties...
    }

    @Override
    public String toString() { ❸
        // Return a stringified message...
    }

    public CustomExchange getExchange() { ❹
        return (CustomExchange)super.getExchange();
    }

    @Override
    public CustomMessage newInstance() { ❺
        return new CustomMessage( ... );
    }

    @Override
    protected Object createBody() { ❻
        // Return message body (lazy creation).
    }

    @Override
   protected void populateInitialHeaders(Map<String, Object>
 map) { ❼
        // Initialize headers from underlying message (lazy
creation).
    }

    @Override
    protected void populateInitialAttachments(Map<String, Da
taHandler> map) { ❽
        // Initialize attachments from underlying message
```

```
(lazy creation).
    }
}
```

❶ Implements a custom message class, *CustomMessage*, by extending the `org.apache.camel.impl.DefaultMessage` class.

❷ Typically, you need a default constructor that creates a message with default properties.

❸ Override the `toString()` method to customize message stringification.

❹ *(Optional)* This is a convenient method that returns a reference to the parent exchange instance, casts it to the correct type.

❺ The `newInstance()` method is called from inside the `MessageSupport.copy()` method. Customization of the `newInstance()` method should focus on copying all of the *custom* properties of the current message instance into the new message instance. The `MessageSupport.copy()` method copies the generic message properties by calling `copyFrom()`.

❻ The `createBody()` method works in conjunction with the `MessageSupport.getBody()` method to implement lazy access to the message body. By default, the message body is `null`. It is only when the application code tries to access the body (by calling `getBody()`), that the body should be created. The `MessageSupport.getBody()` automatically calls `createBody()`, when the message body is accessed for the first time.

❼ The `populateInitialHeaders()` method works in conjunction with the header getter and setter methods to implement lazy access to the message headers. This method parses the message to extract any message headers and inserts them into the hash map, `map`. The `populateInitialHeaders()` method is automatically called when a user attempts to access a header (or headers) for the first time (by calling `getHeader()`, `getHeaders()`, `setHeader()`, or `setHeaders()`).

❽ The `populateInitialAttachments()` method works in conjunction with the attachment getter and setter methods to implement lazy access to the attachments. This method extracts the message attachments and inserts them into the hash map, `map`. The `populateInitialAttachments()` method is automatically called when a user attempts to access an attachment (or attachments) for the first

time by calling `getAttachment()`, `getAttachments()`, `getAttachmentNames()`, or `addAttachment()`.

Index

Symbols

A

C

D

E

I

M

P

S

T

U