PROGRESS SOFTWARE

Artix ESB

Java Router, Programmer's Guide

Version 5.5
December 2008

Java Router, Programmer's Guide

Progress Software

Version 5.5

Published 10 Dec 2008

Legal Notices

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

Open Source Project Resources

Apache Incubator CXF	**Web site:** http://cxf.apache.org/ **User's list:** <user@cxf.apache.org>
Apache Tomcat	**Web site:** http://tomcat.apache.org/ **User's list:** <users@tomcat.apache.org>
	Web site: http://activemq.apache.org/ **User's list:** <users@activemq.apache.org>
Apache Camel	**Web site:** http://activemq.apache.org/camel/enterprise-integration-patterns.html **User's list:** <camel-user@activemq.apache.org>
	Web site: http://servicemix.apache.org **User's list:** <users@servicemix.apache.org>

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`fixed width`	Fixed width (Courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `javax.xml.ws.Endpoint` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `import java.util.logging.Logger;`
`Fixed width italic`	Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`YourUserName`*
Italic	Italic words in normal text represent *emphasis* and introduce *new terms*.
Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the **User Preferences** dialog.

Keying conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
`%`	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
`#`	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
`>`	The notation > represents the MS-DOS or Windows command prompt.
`. . .`	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
`[ ]`	Brackets enclose optional items in format and syntax descriptions.
`{ }`	Braces enclose a list from which you must choose an item in format and syntax descriptions.

|	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in { } (braces).

Admonition conventions

This book uses the following conventions for admonitions:

	Notes display information that may be useful, but not critical.
	Tips provide hints about completing a task or using a tool. They may also provide information about workarounds to possible problems.
	Important notes display information that is critical to the task at hand.
	Cautions display information about likely errors that can be encountered. These errors are unlikely to cause damage to your data or your systems.
	Warnings display information about errors that may cause damage to your systems. Possible damage from these errors include system failures and loss of data.

Understanding Message Formats

Before you can start to program effectively with Java Router, you need to have a clear understanding of how messages and message exchanges are modelled. Because Java Router needs the capability to process many different kinds of message format, the basic message type is designed to have an abstract format. Various programming APIs are provided, however, that enable you to access and transform the data formats that underly message bodies and message headers.

Exchanges

Overview

Exchange objects provide the primary means of accessing messages in Java Router: an exchange object is effectively a wrapper that encapsulates a set of related messages. For example, you can access *In*, *Out*, and *Fault* messages using the `getIn()`, `getOut()`, and `getFault()` accessors defined on `Exchange`. An important feature of exchanges in Java 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. Exchange Object Passing through a Route

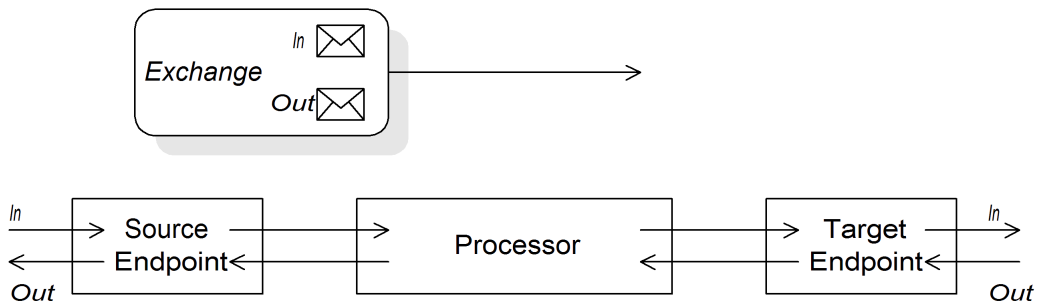


Figure 1 on page 16 shows an exchange object passing through a route. In the context of a route, an exchange object gets passed as the argument of the `Processor.process()` method, which means that the exchange object is directly accessible to the source endpoint, the target endpoint, and all of the 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 on page 16.

Example 1. Exchange Methods

```
// Java
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 116.

Lazy creation of messages

Java 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 would create 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, which could 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 [http://java.sun.com/javaee/5/docs/api/javax/activation/DataHandler.html] (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 also, possibly, using the marshalling and unmarshalling mechanism).

One particularly important feature of Java 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 2 on page 18.

Example 2. Message Interface

```
// Java
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
```

```
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 126.

Lazy creation of bodies, headers, and attachments

Java 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 the moment 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 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 avoid parsing altogether: for example, if a route contains no explicit references to message headers, a message could traverse the route without parsing the headers at all.

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 an expensive operation. If the body is left in the form of a raw buffer, it is probably not an expensive operation; on the other hand, parsing headers always imposes a bit of an overhead. For details about implementing a message type that supports lazy creation, see Implementing the Message Interface on page 129.

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 will remain unparsed until it is accessed explicitly by the application. Most Java Router components would 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, however, components usually rely on *type converters* or *marshalling processors*.

Type converters

Normally, it does not matter very much 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 23). There are various methods in the Java Router API that expose type conversion functionality. For example, the `convertBodyTo(Class type)` method can be inserted into a route in order 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. As a matter of fact, it is not necessary to convert the message body explicitly in this example. You could also write simply:

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

Where the `append()` method automatically converts the message body to a string format 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)`—return the message body as type, `T`.

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

For the complete list of supported conversion types, see Built-In Type Converters on page 23.

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 considerably more expensive than the simple conversions, because it involves parsing the entire message and 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 necessarily have narrower applicability than the simpler conversions: not every message body conforms to an XML structure, so you have to take into account 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

In general, *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 following route definition:

```
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 the *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 applies to *Out* messages that arrive back at the source endpoint). This conversion is usually performed implicitly, using the Java Router type converter. Typically, this involves converting from a low-level format to another low-level format. For example, 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. While the conversions described here are built into the Java Router core, it is also possible to extend the type conversion with custom converters (see *Type Converters* on page 41).

Usually, the type converter is called indirectly 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 follows:

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

Basic type converters

Java Router provides built-in type converters to 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`

Not all conversions amongst these types are supported, however. The built-in converter is focused mainly on providing conversions from the `File` and `String` types. The `File` type can be converted to any of the preceding types,

apart from `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.

Collection type converters

Java Router provides built-in type converters to 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

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

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

In addition to converting amongst themselves, the preceding map types can also be converted into a set, of `java.util.Set` type, where the set elements are of `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 [http://www.saxproject.org/] for details). You can convert to `SAXSource` from the following types: `String`, `InputStream`, `Source`, `StreamSource`, and `DOMSource`.

Custom type converters

Java Router also enables you to implement your own custom type converters. For details of how to implement a custom type converter, see *Type Converters* on page 41.

Implementing a Processor

Java Router allows you to implement a custom processor, which you can then insert into a route in order to perform operations on exchange objects as they pass through the route.

Processing Models

Overview

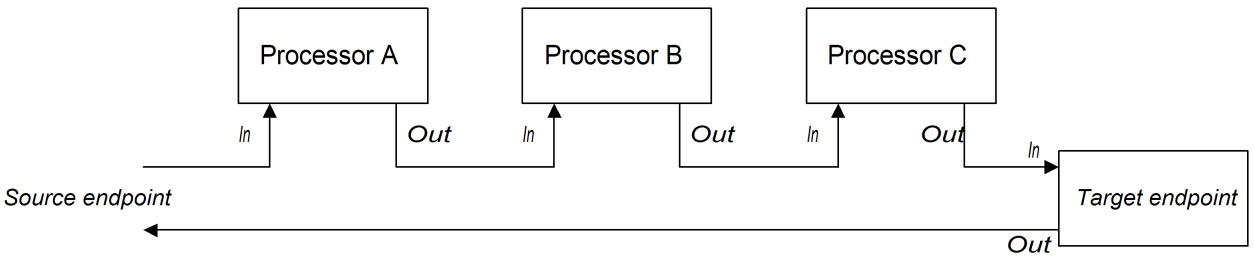
Before you start to implement a processor, you need to consider how the processor is meant to fit into a Java Router route. The most important processing models are, as follows:

- Pipelining model on page 28.
- Chaining model on page 29.

Pipelining model

The *pipelining model* describes the way in which processors are arranged in Pipes and Filters in the *Implementing Enterprise Integration Patterns*. This 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 on page 28.

Figure 2. 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 (for example, using the CXF component).

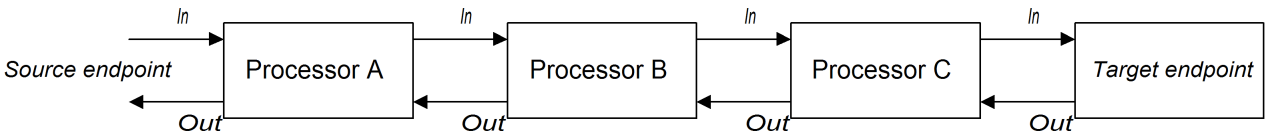
For example, the following Java DSL route shows an example of a pipeline constructed from a sequence of two processors, `ProcessorA`, `ProcessorB`, and a producer endpoint, *`TargetURI`*:

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

Chaining model

The *chaining model* describes an alternative model for arranging processors in a route. In this model, the processors are arranged in a linked list or *chain*, where each processor calls the `process()` method of the next processor in the chain. When the processors are arranged in this way, the exchange's *In* and *Out* messages are normally processed as shown in Figure 3 on page 29.

Figure 3. Chaining Model



Because each processor processes the *In* message *before* delegating the exchange to the next node, the *In* message gets processed in the order shown in Figure 3 on page 29 (left to right). In addition, because each processor process the *Out* message after delegating the exchange to the next node, the *Out* message gets processed in the reverse order (right to left).

Implementing a Simple Processor

Overview

If you need to write code that executes *before* an exchange is delegated to the next processor, you can implement a *simple processor*, as explained in this section. This kind of processor is suitable for use in a pipeline route.

Processor interface

Example 3 on page 30 shows the definition of the `org.apache.camel.Processor` interface, which must be implemented by a simple processor. The interface defines a single method, `process()`, which processes the exchange object.

Example 3. Processor Interface

```
// Java
package org.apache.camel;

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

Implementing the Processor interface

You implement a simple processor by inheriting from `org.apache.camel.Processor` and implementing the `process()` method.

Example 4 on page 30 shows the outline of a simple processor implementation.

Example 4. Simple Processor Implementation

```
// Java
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.
        ...
    }
}
```

Where all of the code in the body of the `process()` method gets executed *before* the exchange object is delegated to the next processor in the chain. Typically, this means that you *cannot* access the reply (if any) from the endpoint of the route, because the exchange object does not reach the end of the route until after the exchange is delegated to the next processor in the route. This limitation can be overcome by implementing a delegate processor instead—see Implementing a Delegate Processor on page 32.

For examples of how to access the message body and header values inside a simple processor, see Accessing Message Content on page 35.

Inserting the simple processor into a route

To insert a simple processor into a route, use the `process()` DSL command.

Create an instance of your custom processor and then pass this instance as an argument to the `process()` method, as follows:

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

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

Implementing a Delegate Processor

Overview

If you need to write code that executes both before and after an exchange is delegated to the next processor, you can implement a *delegate processor*, as explained in this section. Delegate processors conform to the chaining model for building routes.

DelegateProcessor class

Example 5 on page 32 shows a partial outline of the `org.apache.camel.processor.DelegateProcessor` class. The main difference between the `DelegateProcessor` class and the `Processor` class is that the `DelegateProcessor` class has a bean property, `processor`, which holds a reference to the next processor in the chain. This makes it possible for you to call the next processor explicitly when you write the code for the `process()` method. The most convenient way to call the next processor in the chain is to call the `processNext()` method.

Example 5. DelegateProcessor Class

```
package org.apache.camel.processor;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.impl.ServiceSupport;
import org.apache.camel.spi.Policy;
import org.apache.camel.util.ServiceHelper;

public class DelegateProcessor extends ServiceSupport imple
ments Processor {
    protected Processor processor;

    public DelegateProcessor() {
    }

    public Processor getProcessor() {
        return processor;
    }

    public void setProcessor(Processor processor) {
        this.processor = processor;
    }
    ...
    public void process(Exchange exchange) throws Exception
{
        processNext(exchange);
```

```
    }
    ...
    protected void processNext(Exchange exchange) throws Ex
ception {
        if (processor != null) {
            processor.process(exchange);
        }
    }
}
```

Where the `setProcessor()` method enables the route builder to inject a reference to the next processor in the chain and the `process()` method must be overridden by your custom delegate processor class.

Extending the DelegateProcessor class

You implement a delegate processor by extending `DelegateProcessor` and implementing the `process()` method. Example 6 on page 33 shows the outline of a delegate processor implementation.

Example 6. Delegate Processor Implementation

```
// Java
import org.apache.camel.processor.DelegateProcessor;

public class MyDelegateProcessor extends DelegateProcessor {
    public MyProcessor() { }

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

        // to the next processor in the chain.
        ...
        processNext(exchange);

        // Insert code that gets executed *after* delegating
        // to the next processor in the chain.
        ...
    }
}
```

Where the `process()` method contains code that gets executed *before* delegating to the next processor in the chain, as well as code that gets executed *after* delegating. If the processors in your route are chained according to the chaining model (see Chaining model on page 29), this means that you

can access request messages before the call to `processNext()` and access reply messages after the call.

For examples of how to access the message body and header values inside a simple processor, see Accessing Message Content on page 35.

Inserting the delegate processor into a route

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

```
// Java
org.apache.camel.processor.DelegateProcessor myProc = new
MyDelegateProcessor();

from("SourceURL").intercept(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, first of all obtain 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 7 on page 35 shows an example of a custom processor that access the value of a header named `Authorization` (which, for example, might represent HTTP Basic Authentication credentials). This example uses the `ExchangeHelper.getMandatoryHeader()` method, which saves you having to test for a null header value.

Example 7. Accessing an Authorization Header

```
// Java
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 18.

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 could use the processor shown in Example 8 on page 35.

Example 8. Accessing the Message Body

```
// Java
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 2 on page 18 for more details.

The ExchangeHelper Class

Overview

The org.apache.camel.util.ExchangeHelper [http://activemq.apache.org/camel/maven/camel-core/apidocs/org/apache/camel/util/ExchangeHelper.html] class is a Java Router utility class that provides methods that typically come in useful when implementing a processor.

Resolve an endpoint

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

```
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. For example, given an exchange instance, `exchange`, you could create a new file endpoint as follows:

```
// Java
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 essential difference between them is that the original `get`*`BeanProperty`*`()` accessors return `null`, if the corresponding property is unavailable, whereas the `getMandatory`*`BeanProperty`*`()` wrapper methods throw a Java exception. The following wrapper methods are implemented in `ExchangeHelper`:

```
public final class ExchangeHelper {
    ...
   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

There are several different exchange patterns for which an exchange object is capable of holding an *In* message. Likewise, several different exchange patterns are 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 quickly by calling `ExchangeHelper.getContentType(exchange)`. To implement this, the

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

Type Converters

Java Router has a built-in type conversion mechanism, which is mainly used for the purpose of converting 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 need to understand, if you are going to write a custom type converter. If all you want to do is use the build-in type converters, see *Understanding Message Formats* on page 15 instead.

TypeConverter interface

Example 9 on page 42 shows the definition of the `org.apache.camel.TypeConverter` interface, which all type converter classes must implement.

Example 9. TypeConverter Interface

```
// Java
package org.apache.camel;

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

Master type converter

The Java 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. That is, 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 the slave.

For users of the type conversion mechanism, the master type converter is the most important. It provides the entry point for accessing the conversion mechanism. While starting up, Java Router automatically associates a master type converter instance with the `CamelContext` object. Hence, to obtain a reference to the master type converter, you can call the `CamelContext.getTypeConverter()` method. For example, if you have an exchange object, `exchange`, you could obtain a reference to the master type converter as follows:

```
// Java
org.apache.camel.TypeConverter tc = exchange.getContext().get
TypeConverter();
```

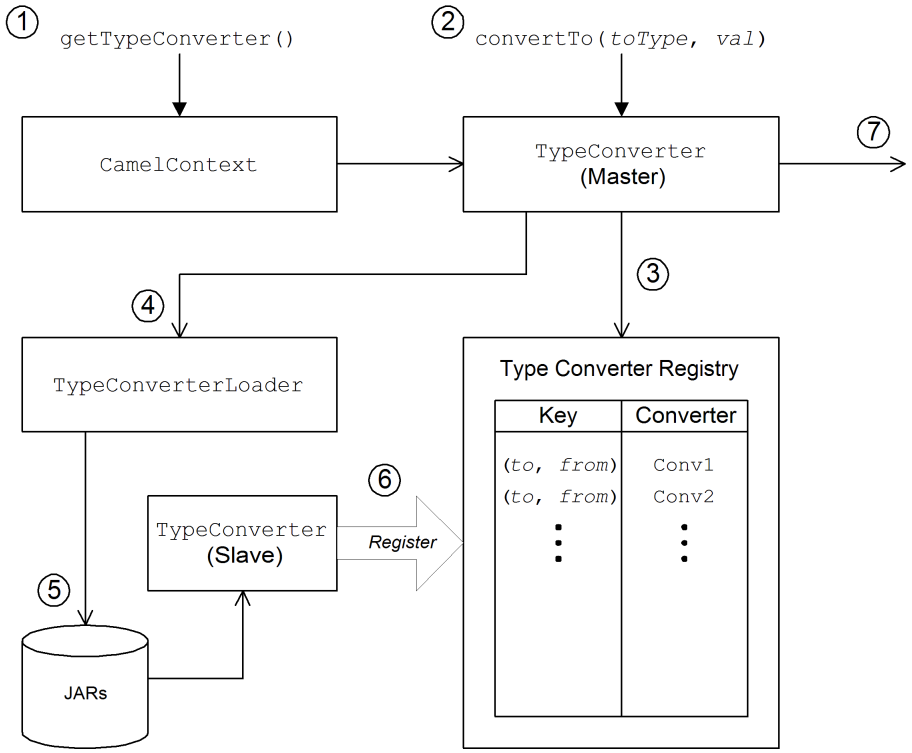
Type converter loader

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. In practice, Java Router currently uses only one kind of type converter loader, the *annotation type converter loader* (of `AnnotationTypeConverterLoader` type).

Type conversion process

Figure 4 on page 44 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 4. Type Conversion Process



Type conversion steps

The type conversion mechanism proceeds as follows:

1. The `CamelContext` object holds a reference to the master `TypeConverter` instance. Normally, 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 `convertTo()` on the master type converter. This method requests 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 just a front end for many different slave type converters, it tries to find 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 resorts to loading a new type converter, using the type converter loader.

5. The type converter loader searches the available JAR libraries on the classpath in order 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 47).

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. The converted data value is returned or `null`, if the conversion does not succeed.

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 Java 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 on page 46.
2. Create a TypeConverter file on page 47.
3. Package the type converter on page 47.

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 10 on page 46 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 10. Example of an Annotated Converter Class

```
// Java
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));
    }
}
```

Where 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 like. What matters 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

Normally, you package the type converter 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 Java Router application.

Implementing a Component

This chapter provides a general overview of the approaches you can use to implement a Java Router component.

Component Architecture

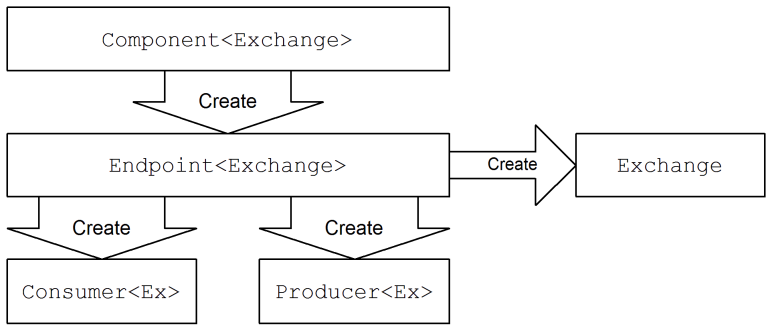
Factory Patterns for a Component

Overview

A Java 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 acts as factories for creating `Consumer`, `Producer`, and `Exchange` objects. These relationships are summarized in

Figure 5 on page 51

Figure 5. Component Factory Patterns



Component

A component implementation is essentially an endpoint factory. Hence, 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 as follows: `file://tmp/messages/input`. When you install a new component in Java 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. So, every time Java Router encounters a new endpoint URI, it creates a new endpoint instance.

The class that implements an endpoint must inherit from the `org.apache.camel.Endpoint` interface. The `Endpoint` interface defines the following factory methods:

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

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

Consumer

A consumer endpoint always appears at the start of a route and it encapsulates the code responsible for receiving incoming requests and dispatching outgoing replies (that is, it *consumes* requests). Another way of expressing this is to say that a consumer represents a *service*.

An implementation of a consumer class must inherit from the `org.apache.camel.Consumer` interface. There are, in fact, a number of different patterns you can follow when implementing a consumer class, as is described in detail in Consumer Patterns on page 56.

Producer

A producer endpoint always appears at the end of a route and it encapsulates the code responsible for dispatching outgoing requests and receiving incoming replies (it *produces* requests). Expressed in the terminology of a Service Oriented Architecture, the producer could also be identified as a *service consumer* (beware of the potential for confusion, however, with the term *consumer* as it is used in Java Router).

An implementation of a producer class must inherit from the `org.apache.camel.Producer` interface. If you want, you can optionally implement the producer to support an asynchronous style of processing—see Asynchronous Processing on page 60 for details.

Exchange

An exchange object encapsulates 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.

An implementation of an exchange class must inherit from the `org.apache.camel.Exchange` interface. Often a component implementation can simply use the default implementation, `DefaultExchange`. Sometimes it can be useful to customize the exchange implementation—for example, if you want to associate some extra properties or data with the exchange object.

Message

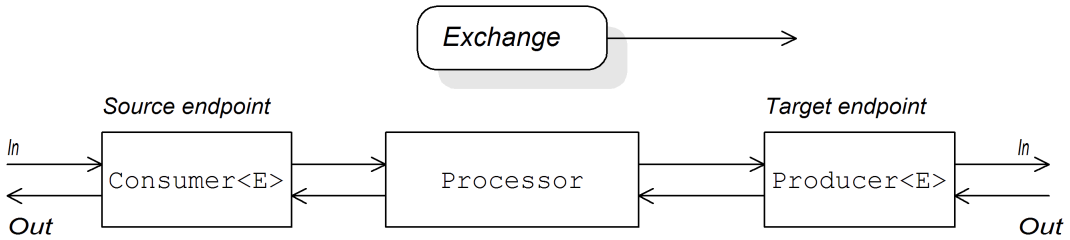
There are three different kinds of messages, *In* messages, *Out* messages, and *Fault* messages, all of which are represented by the same message type, `org.apache.camel.Message`. You do not always need to customize the message implementation—the default implementation, `DefaultMessage`, is often adequate.

Using a Component in a Route

Overview

A Java Router route is essentially a chain 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 chain is illustrated in Figure 6 on page 54.

Figure 6. 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, Java 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 51.

Processors

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

Target endpoint

At the end of the route you have the target endpoint, which is represented by an `org.apache.camel.Producer` object. Because it comes at the end of a processor chain, 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, Java Router creates the appropriate `Producer` type based on the component prefix from the endpoint URI.

Consumer Patterns

Overview

As a consequence of its position at the start of a route, the consumer plays an especially important role. Many important features of the route are determined by the consumer. For example, the consumer gets to determine the threading model for processing the exchanges that pass through the route. The consumer is also responsible for determining the format of incoming request messages.

Threading

In order to accommodate different kinds of threading models for processing incoming requests, Java Router supports a variety of different consumer implementation patterns: the *event-driven* pattern allows the consumer to be driven by an external thread; the *scheduled poll* pattern creates a dedicated thread pool to drive the consumer; and the *polling* pattern leaves the threading model undefined.

Alternative consumer patterns

You can implement a consumer based on one of the following patterns:

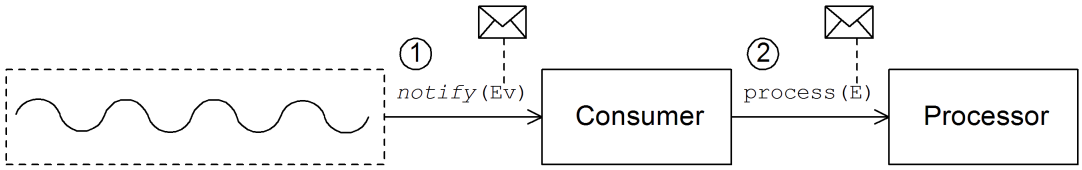
- Event-driven pattern on page 56.
- Scheduled poll pattern on page 57.
- Polling pattern on page 58.

Event-driven pattern

In the event-driven pattern, 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 Java Router JMX component, where events are initiated by the JMX library, which calls the `handleNotification()` method to initiate request processing—see Example 22 on page 100 for details.

Figure 7 on page 57 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 7. 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 the figure, this is represented by the `notify()` method). The thread that calls `notify()` is normally a separate part of the application. Hence, 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 in order to receive notifications from JMX. The threads that drive the consumer processing are created within the JMX layer.

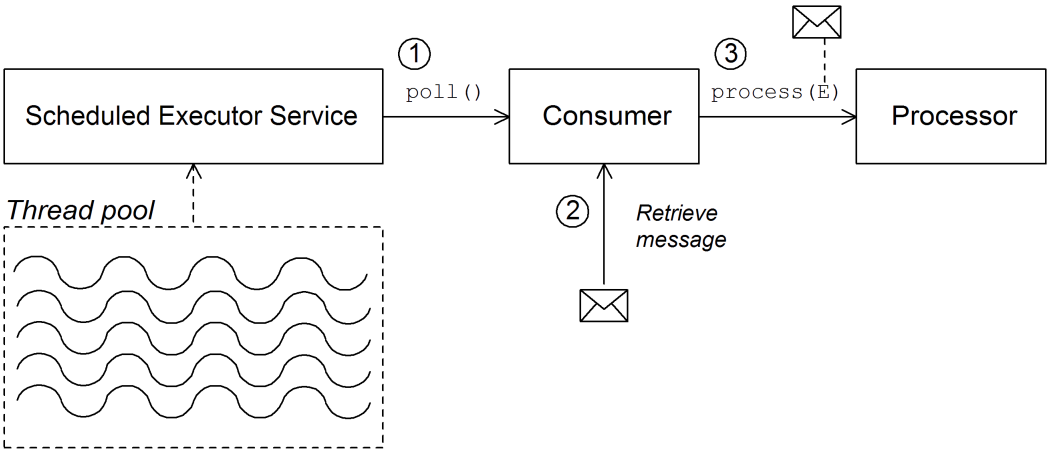
2. In the body of the `notify()` method, the consumer first converts the incoming event into an exchange object, `E`, and then calls `process()` on the next processor in the route, passing the exchange object as its argument.

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 is capable of executing a particular task at timed intervals and it also manages a pool of threads, which it uses to run the task instances.

Figure 8 on page 58 shows an outline of the scheduled poll consumer pattern.

Figure 8. 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, which it can use to initiate consumer processing. After each scheduled time interval has elapsed, the scheduled executor service tries to get hold of a free thread from its pool (there are five threads in the pool by default). If a free thread is available, it uses the 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 should attempt to retrieve an incoming message. If no request is available, the `poll()` method should return right away.

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.

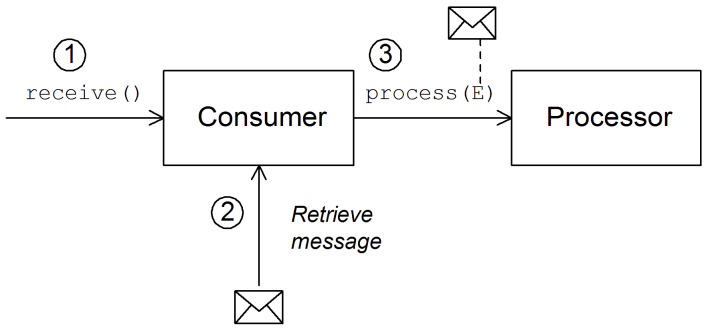
Polling pattern

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()`, and `receive(long timeout)`. In general, 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 9 on page 59 shows an outline of the polling consumer pattern.

Figure 9. 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 (`receive()`, `receiveNoWait()`, or `receive(long timeout)`) are 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: if the method is `receiveNoWait()`, return immediately; if the method is `receive(long timeout)`, wait for the specified timeout (usually specified in milliseconds) before returning; and if the method is `receive()`, wait until a message is received (possibly indefinitely).

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.

Asynchronous Processing

Overview

Producer endpoints normally follow a *synchronous* pattern when processing an exchange. That is, when the preceding processor in a chain 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.

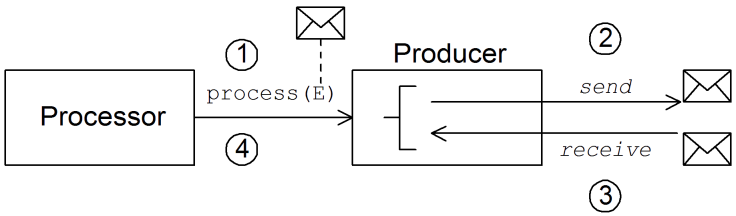
Sometimes, however, you might prefer to decouple the preceding processor from the producer, so that the processor's thread is freed up immediately and the `process()` call does *not* block. In this case, you should implement the producer using an *asynchronous* pattern, which gives the preceding processor 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 asynchronous patterns for implementing a producer endpoint.

Synchronous producer

Figure 10 on page 60 shows an outline of a synchronous producer, where the preceding processor blocks until the producer has finished processing the exchange.

Figure 10. Synchronous Producer



The synchronous producer processes an exchange as follows:

1. The preceding processor in the chain 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.

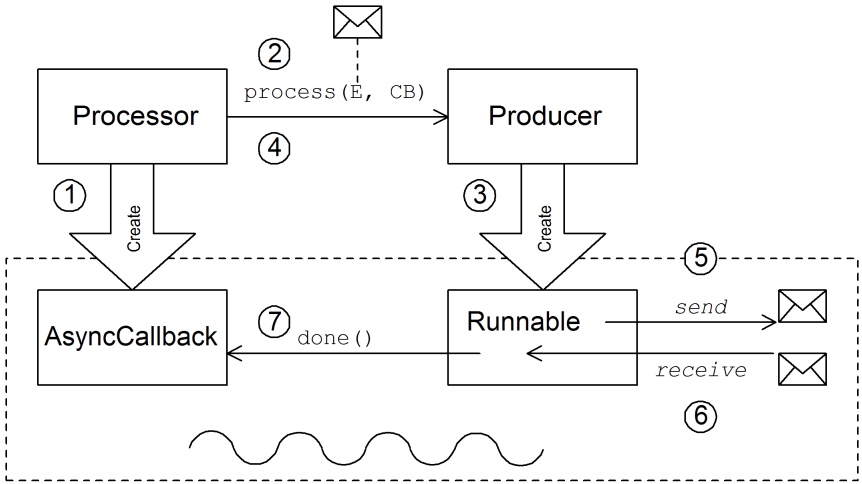
3. If required by the exchange pattern, the producer waits for the reply (*Out* or *Fault* message) to arrive from the endpoint. Potentially, this step could cause the `process()` method to block indefinitely. If the exchange pattern does not mandate a reply, however, the `process()` method could return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

4. When the `process()` method returns (potentially after having been blocked for some time), the exchange object contains the reply from the synchronous call (either an *Out* message or a *Fault* message).

Asynchronous producer

Figure 11 on page 61 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 11. 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 leg 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 and 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.
5. Processing of the exchange now takes place in the separate sub-thread. First of all, 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 Java Router custom component.

Which interfaces do you need to implement?

When implementing a component, it is almost always 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 is sometimes also necessary to implement the following Java interfaces:

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

Implementation steps

In outline, you would typically implement a custom component as follows:

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

   See *Component Interface* on page 73.

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

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

   - `createProducer()`.

- `createConsumer()`.

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

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

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

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

See *Endpoint Interface* on page 81.

3. *Implement the Consumer interface*—there are several different approaches you can take to implementing a consumer, depending on whether you need to implement an event-driven pattern, a scheduled poll pattern, or a polling pattern. 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 100.

4. *Implement the Producer interface*—to implement a producer, derive from the `DefaultProducer` class and implement the `process()` method.

   See *Producer Interface* on page 107.

5. *(Optionally) Implement Exchange or Message interfaces*—frequently, the default implementations of `Exchange` and `Message` can be used directly. Occasionally, you might find it necessary to customize these types.

   See *Exchange Interface* on page 115 and *Message Interface* on page 125.

Installing and configuring the component

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

- *Add the component directly to the CamelContext*—use the `CamelContext.addComponent()` method to add a component programatically. For more details, see Adding Components to the Camel Context in the *Deployment Guide*.
- *Add the component using Spring configuration*—use the standard Spring `bean` element to create a component instance. The bean's `id` attribute implicitly defines the component prefix. For details, see Configuring a Component on page 69.
- *Configure Java Router to auto-discover the component*—using auto-discovery, you can ensure that Java Router automatically loads the component on demand. For details, see Setting Up Auto-Discovery on page 67.

Auto-Discovery and Configuration

Setting Up Auto-Discovery

Overview

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

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 it 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 to this file.

Example

For example, you could enable auto-discovery for the Java 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

Alternatively, you can add a component by configuring it in the Java 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, Java Router instantiates the referenced class and injects the properties specified in the Spring configuration.

Note

This mechanism has priority over auto-discovery. That is, if the CamelContext can find 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 would like to inject into your component class, define them as bean properties. For example:

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

Where `getProperty()` and `setProperty()` 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 11 on page 69.

Example 11. 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 ht
tp://www.springframework.org/schema/beans/spring-beans-2.0.xsd

       http://activemq.apache.org/camel/schema/spring ht
```

```
tp://activemq.apache.org/camel/schema/spring/camel-spring.xsd">


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

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

Where 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 12 on page 70 shows an example of how to configure the Java Router 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 12. JMS Component Configuration in camel-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans ht
tp://www.springframework.org/schema/beans/spring-beans-2.0.xsd

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


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

  <bean id="jms" class="org.apache.camel.component.jms.JmsCom
ponent"> ❷
    <property name="connectionFactory"> ❸
      <bean class="org.apache.activemq.ActiveMQConnectionFact
```

```
ory">
         <property name="brokerURL"
                   value="vm://localhost?broker.persist
ent=false&amp;broker.useJmx=false"/> ❹
       </bean>
    </property>
  </bean>
</beans>
```

❶ The `CamelContext` will automatically instantiate 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`, Java Router would automatically load the JMS component using the settings from the `jms` bean element.

❸ JMS is just a wrapper for a messaging service. You need to 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` (meaning that messages in the broker are not stored persistently) and `broker.useJmx=false` (meaning that the broker does not open a JMX port).

Component Interface

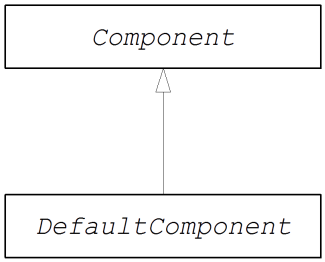
This chapter describes in detail how to implement the `Component` *interface.*

The Component Interface

Overview

To implement a Java 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 12 on page 74 shows the relevant Java interfaces and classes that make up the `Component` inheritance hierarchy.

Figure 12. Component Inheritance Hierarchy



The Component interface

Example 13 on page 74 shows the definition of the `org.apache.camel.Component` interface.

Example 13. Component Interface

```
// Java
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()`—reference the `CamelContext` to which this `Component` belongs. The `setCamelContext()` method is automatically called when you add the component to a `CamelContext`.

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

Implementing the Component Interface

The DefaultComponent class

Normally, 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:

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

Where `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.

To see how URI parsing works in practice, consider the following sample endpoint URI for the `file` component:

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

For this URI, the following arguments would be 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`

Header 1	Header 2
`parameters`	Two entries are set in `java.util.Map`: parameter `delete` is boolean `true`, and parameter `moveNamePostfix` has the string value, `.old`.

Parameter injection

You can use the parameters extracted from the URI query options to perform parameter injection on the endpoint's bean properties. The `DefaultComponent` class provides a helper method, `setProperties()`, that performs the parameter injection for you.

For example, imagine that you want to define a custom endpoint that supports two URI query options: `delete` and `moveNamePostfix`. First of all, you need to define the corresponding bean methods (getters and setters) in the endpoint class:

```
// Java
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;
    }
}
```

Then, in the implementation of `createEndpoint()`, call `setProperties(Object bean, Map parameters)`, passing the endpoint instance as the `bean` argument and passing the URI query options as the `parameters` argument (see Example 14 on page 78). This is all you need to do in order to ensure that URI query options get injected into your custom endpoint instance.

It is also possible to inject URI query options into *consumer* parameters. For details, see Consumer parameter injection on page 95.

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 intended to be 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 (implementing the `java.util.concurrent.ScheduledExecutorService` interface). The `ScheduledThreadPoolExecutor` instance is initialized with a thread pool of fixed size, 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()`.

Creating an endpoint

Example 14 on page 78 outlines how to implement the `DefaultComponent.createEndpoint()` method, which is responsible for creating endpoint instances on demand.

Example 14. Implementation of createEndpoint()

```
// Java
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
        setProperties(result, parameters); ❹
        // ...
        return result;
    }
}
```

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

❷ When inheriting from `DefaultComponent`, you must implement the `createEndpoint()` method with three arguments (see URI parsing on page 76).

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

❹ The `setProperties()` method is defined in `DefaultComponent` and is responsible for performing parameter injection on the endpoint instance. It uses introspection (Java reflection) to identify each *CustomEndpoint* bean parameter that matches a corresponding parameter name and then calls the relevant setter method to inject the parameter value.

Example

Example 15 on page 79 shows the complete implementation of the `FileComponent` class, which is taken from the Java Router file component implementation.

Example 15. FileComponent Implementation

```
// Java
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<FileEx
change> {
    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);
        setProperties(result, parameters);
        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 14 on page 78. The implementation creates an instance of a file endpoint (of `FileEndpoint` type) and then injects the URI query options by calling `setProperties()`.

Endpoint Interface

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

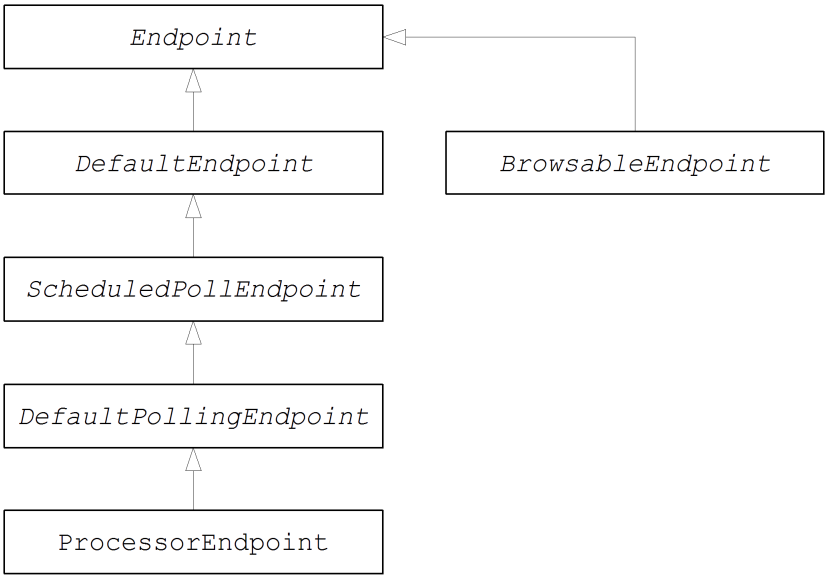
The Endpoint Interface

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. Three different approaches to implementing an endpoint are described here: event-driven, scheduled poll, and polling. These endpoint implementation patterns complement the corresponding patterns for implementing a consumer—see Implementing the Consumer Interface on page 100.

Figure 13 on page 82 shows the relevant Java interfaces and classes that make up the `Endpoint` inheritance hierarchy.

Figure 13. Endpoint Inheritance Hierarchy



The Endpoint interface

Example 16 on page 83 shows the definition of the `org.apache.camel.Endpoint` interface.

Example 16. Endpoint Interface

```
// Java
package org.apache.camel;

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

    String getEndpointUri();

    CamelContext getContext();

    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()`—return `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 same (that is, 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. That is, each time you refer to the URI in a route, a new endpoint instance would be created.
- `getEndpointUri`—return the endpoint URI of this endpoint.
- `getContext()`—return a reference to the `CamelContext` instance to which this endpoint belongs.
- `createExchange()`—is an overloaded method with the following variants:
  - `E createExchange()`—create a new exchange instance with a default exchange pattern setting.

- `E createExchange(ExchangePattern pattern)`—create a new exchange instance with the specified exchange pattern.
- `E createExchange(Exchange exchange)`—convert 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 should copy it into a new instance of the correct type (a default implementation of this method is provided in the `DefaultEndpoint` class).

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

Endpoint singletons

In order to avoid unnecessary overheads, it is a good idea to create just 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 be treated as not the same.

Implementing the Endpoint Interface

Alternative ways of implementing an endpoint

The following alternative endpoint implementation patterns are supported:

- Event-driven endpoint implementation on page 85
- Scheduled poll endpoint implementation on page 87
- Polling endpoint implementation on page 89

Event-driven endpoint implementation

If your custom endpoint conforms to the event-driven pattern (see Consumer Patterns on page 56), implement it by inheriting from the abstract class, `org.apache.camel.impl.DefaultEndpoint`, as shown in Example 17 on page 85.

Example 17. Implementing DefaultEndpoint

```
// Java
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<CustomEx
change> { ❶

    public CustomEndpoint(String endpointUri, Component compon
ent) { 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(getContext(), getExchangePat
tern());
    }

    public CustomExchange createExchange(ExchangePattern pat
tern) {
        return new CustomExchange(getContext(), pattern);
    }
}
```

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

❷ You need 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, in order to create a producer endpoint.

❹ Implement the `createConsumer()` factory method, in order to create an event-driven consumer instance. Do *not* override the `createPollingConsumer()` method.

❺ If you intend to customize the exchange implementation, you should override the `createExchange()` and `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 by default.

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.

- `getContext()`—returns a reference to the `CamelContext`.
- `getComponent()`—returns a reference to the parent component.
- `getExecutorService()`—return a reference to a scheduled executor service (of `java.util.concurrent.ScheduledExecutorService` type).
- `createPollingConsumer()`—creates a polling consumer, whose functionality is based on the event-driven consumer. In other words, 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, which ensures that the method also works for custom exchange types.

Scheduled poll endpoint implementation

If your custom endpoint conforms to the scheduled poll pattern (seeConsumer Patterns on page 56), implement it by inheriting from the abstract class, `org.apache.camel.impl.ScheduledPollEndpoint`, as shown in Example 18 on page 87.

Example 18. ScheduledPollEndpoint Implementation

```
// Java
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<Cus
tomExchange> { 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 CustomProdu
cer(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 pat
tern) {
        return new CustomExchange(getContext(), pattern);
    }
}
```

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

❷ You need 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, in order to create a producer endpoint.

❹ Implement the `createConsumer()` factory method, in order to create a scheduled poll consumer instance. 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 95.

❻ If you intend to customize the exchange implementation, you should override the `createExchange()` and `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 by default.

Polling endpoint implementation

If your custom endpoint conforms to the polling consumer pattern (see Consumer Patterns on page 56), implement it by inheriting from the abstract class, `org.apache.camel.impl.DefaultPollingEndpoint`, as shown in Example 19 on page 89.

Example 19. DefaultPollingEndpoint Implementation

```
// Java
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<Cus
tomExchange> {
    ...
    public PollingConsumer<CustomExchange> createPollingCon
sumer() throws Exception {
        PollingConsumer<CustomExchange> result = new CustomCon
sumer(this);
        configureConsumer(result);
        return result;
    }

    // Do NOT implement createConsumer(). It is already imple
mented 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 104.

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 18 on page 87 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 optionally implement the `org.apache.camel.spi.BrowsableEndpoint` interface, as shown in Example 20 on page 90. It makes sense to implement this interface, if the endpoint performs some sort of buffering of incoming events. For example, the Java Router SEDA endpoint implements the `BrowsableEndpoint` interface—see Example 21 on page 91.

Example 20. BrowsableEndpoint Interface

```
// Java
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 21 on page 91 shows the implementation of `SedaEndpoint`, which is taken from the Java 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 21. SedaEndpoint Implementation

```
// Java
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<Exchange> { ❶
    private BlockingQueue<Exchange> queue;

    public SedaEndpoint(String endpointUri, Component compon
ent, 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, para
meters));
    }

    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 just a single endpoint instance should be created for each unique URI string.

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

Consumer Interface

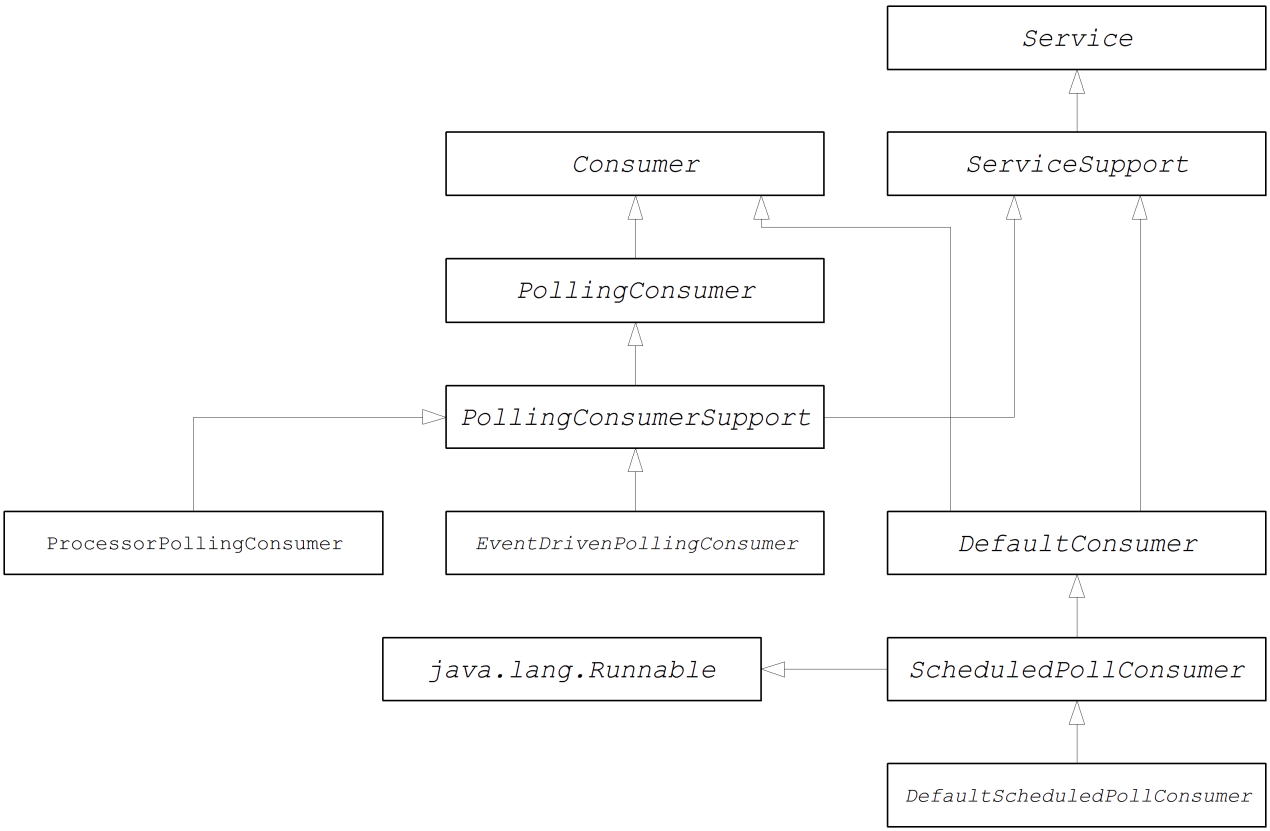
This chapter describes in detail how to implement the `Consumer` interface, which is an essential step in the implementation of a Java Router component.

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 on page 56) and this degree of flexibility is reflected in the inheritance hierarchy (Figure 14 on page 95), which includes several different base classes for implementing a consumer.

Figure 14. Consumer Inheritance Hierarchy



Consumer parameter injection

For consumers that follow the scheduled poll pattern (see Scheduled poll pattern on page 57), Java 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
```

Java Router provides support for automatically injecting query options of the form `consumer.*`. For the `consumer.myConsumerParam` parameter, you

would need to define corresponding setter and getter methods on the `Consumer` implementation class, as follows:

```
// Java
public class CustomConsumer<E extends Exchange> extends Sched
uledPollConsumer<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 87). For example, here is an example of a `createConsumer()` method implementation, taken from the `FileEndpoint` class in the file component:

```
// Java
...
public class FileEndpoint extends ScheduledPollEndpoint<FileEx
change> {
    ...
    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()` in order to inject the consumer parameters (see preceding Java example).
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 1 on page 97 (which can appear as query options in the endpoint URI).

Table 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 periodicity. That is, 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. That is, 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

Java 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:

```
// Java
package org.apache.camel;

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

Where `createConsumer()` returns an event-driven consumer and `createPollingConsumer()` returns a polling consumer. Normally, you would implement only one or other of these methods. For example, if you are following the event-driven pattern for your consumer, you would implement the `createConsumer()` method. But what about the other consumer creation method? One possibility would be to provide a method implementation that simply raises an exception. With the help of the conversion classes, however, Java Router is able to provide a more useful default implementation.

For example, assume you want to implement your consumer according to the event-driven pattern. In this case, you would 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:

```
// Java
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()`, `receive(long timeout)`, and `receiveNoWait()` methods.

Analogously, if you are implementing your consumer according to the polling pattern, you would 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 on page 100
- Scheduled poll consumer implementation on page 102
- Polling consumer implementation on page 104

Event-driven consumer implementation

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

Example 22 on page 100 shows the implementation of the `JMXConsumer` class, which is taken from the Java 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 therefore necessary to implement the `NotificationListener` interface and override the `handleNotification()` method, as shown in Example 22 on page 100.

Example 22. JMXConsumer Implementation

```
// Java
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, it follows that 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 will need to identify an analogous event listener method to implement in your custom consumer.

❹ This line of code combines two steps. First of all, the JMX notification object is converted into an exchange object, which is the generic

representation of an event in Java Router. The newly created exchange object is then 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 on page 56).

Example 23 on page 102 outlines how to implement a consumer that follows the scheduled poll pattern, which is implemented by extending the `ScheduledPollConsumer` class.

Example 23. ScheduledPollConsumer Implementation

```
// Java
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 Sched
uledPollConsumer<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 in order 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 60.

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 24 on page 104 outlines how to implement a consumer that follows the polling pattern, which is implemented by extending the `PollingConsumerSupport` class.

Example 24. PollingConsumerSupport Implementation

```
// Java
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, 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.

Producer Interface

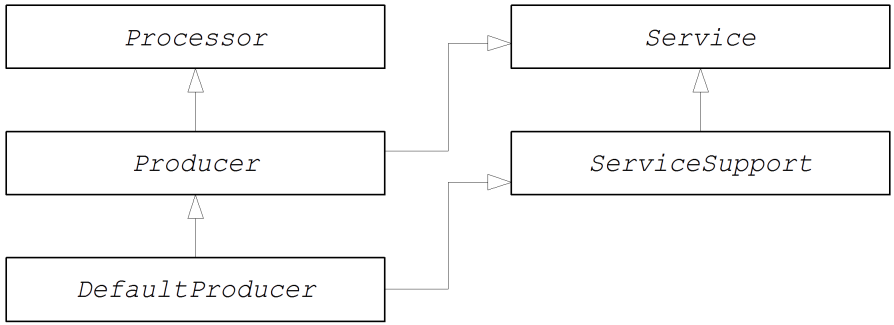
This chapter describes in detail how to implement the `Producer` *interface, which is an essential step in the implementation of a Java Router component.*

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 15 on page 108 shows the inheritance hierarchy for producers.

Figure 15. Producer Inheritance Hierarchy



The Producer interface

Example 25 on page 108 shows the definition of the `org.apache.camel.Producer` interface.

Example 25. Producer Interface

```
// Java
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)*—is the most important method. A producer is essentially a special type of processor that happens to send 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()`—return 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 just 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 could 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 60.

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 26 on page 109.

Example 26. AsyncProcessor Interface

```
// Java
package org.apache.camel;
```

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

Where 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 27 on page 110.

Example 27. AsyncCallback Interface

```
// Java
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 37.

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 on page 111.
- How to implement an asynchronous producer on page 112.

How to implement a synchronous producer

Example 28 on page 111 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) has been received.

Example 28. DefaultProducer Implementation

```
// Java
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, send the *In* message to the designated endpoint.
- If the exchange pattern anticipates the receipt of an *Out* message or a *Fault* message, 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 `exchange.setOut()` or `exchange.setFault()` to attach the reply to the exchange object and then return.

How to implement an asynchronous producer

Example 29 on page 112 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 29. CollectionProducer Implementation

```
// Java
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 a variety of 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 would 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 then call `callback.done()` to notify the caller that processing is complete.

Exchange Interface

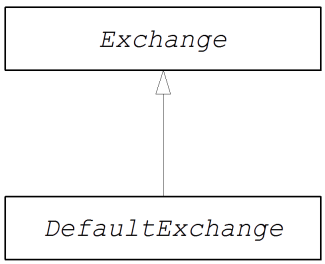
This chapter describes in detail how to implement the `Exchange` interface, which is an optional step in the implementation of a Java Router component.

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 would consist of an *In* message and an *Out* message). Figure 16 on page 116 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 16. Exchange Inheritance Hierarchy



The Exchange interface

Example 30 on page 116 shows the definition of the `org.apache.camel.Exchange` interface.

Example 30. Exchange Interface

```
// Java
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 (that is, if the current *Out* message is `null`, a new message would automatically be created);

  - `getOut(boolean lazyCreate)` with a boolean argument triggers lazy creation, if the argument is `true`, but otherwise returns the current (possibly `null`) value.

- `getFault()`—getter message for the fault message. There are two varieties of `getFault()` method in the `DefaultExchange` class:
  - `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`, but otherwise returns the current (possibly `null`) value.

  The `DefaultExchange` class also defines a `setFault()` method.
- `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()`—return a reference to the associated `CamelContext` instance.
- `newInstance()`—create 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()`—create 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()`—copy the generic contents (apart from the exchange ID) of the specified generic exchange object, `exchange`, into the current exchange instance. Because this method has to 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 needed only for exchanges that can participate in a transaction.
- `setExchangeId()`, `getExchangeId()`—getter and setter methods for the exchange ID. It is an implementation detail, whether or not you need to use an exchange ID in your custom component.

Implementing the Exchange Interface

How to implement a custom exchange

Example 31 on page 121 outlines how to implement an exchange by extending the `DefaultExchange` class.

Example 31. Custom Exchange Implementation

```
// Java
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 Implement 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` (defined in the `DefaultExchange` constructor).

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

❺ The `newInstance()` method is called from inside the `DefaultExchange.copy()` method. Your 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 takes care of copying the generic exchange properties (by calling `copyFrom()`).

❻ *(Optional)* Needed only if you implement a custom message type. The `createInMessage()`, `createOutMessage()`, and `createFaultMessage()` methods are implemented in order to support lazy message creation when you are using a custom message type, *CustomMessage*. For example, if you want to lazily create an *In* message by calling `getIn()`, you would implement `createInMessage()` to ensure that a message of type, *CustomMessage*, is created (`DefaultExchange.getIn()` 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 32 on page 123 shows the implementation of the `FileExchange` class, which is taken from the Java 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, and the only supported exchange pattern is `InOnly`.

Example 32. FileExchange Implementation

```
// Java
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. In particular, the form of constructor called here ensures that the `file` property gets copied into the new instance.

Message Interface

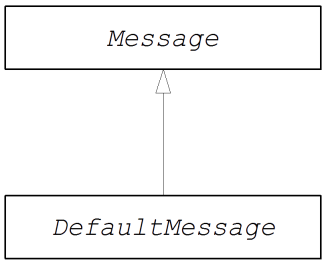
This chapter describes in detail how to implement the `Message` interface, which is an optional step in the implementation of a Java Router component.

The Message Interface

Overview

An instance of `org.apache.camel.Message` type can represent any kind of message (*In*, *Out*, or *Fault*). Figure 17 on page 126 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 17. Message Inheritance Hierarchy



The Message interface

Example 33 on page 126 shows the definition of the `org.apache.camel.Message` interface.

Example 33. Message Interface

```
// Java
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. It is an implementation detail, whether or not you need to use a message ID in your custom component.
- `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()`—create a new, identical (including the message ID) copy of the current custom message object.
- `copyFrom()`—copy the complete contents (including the message ID) of the specified generic message object, `message`, into the current message instance. Because this method has to 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 34 on page 129 outlines how to implement a message by extending the `DefaultMessage` class.

Example 34. Custom Message Implementation

```
// Java
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 Implement 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 in order to customize message stringification.

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

❺ The `newInstance()` method is called from inside the `MessageSupport.copy()` method. Your 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 takes care of copying 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 should parse the message to extract any message headers and insert them into the hash map, `map`. The `populateInitialHeaders()` method will automatically be 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 should extract the message attachments

and insert them into the hash map, `map`. The `populateInitialAttachments()` method will automatically be called when a user attempts to access an attachment (or attachments) for the first time (by calling `getAttachment()`, `getAttachments()`, `getAttachmentNames()`, or `addAttachment()`).