Red Hat JBoss Fuse 6.0

Programming EIP Components

Using the Apache Camel API to create better routes

Last Updated: 2017-10-13

Red Hat JBoss Fuse 6.0 Programming EIP Components

Using the Apache Camel API to create better routes

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Copyright © 2013 Red Hat.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide describes how to use the Apache Camel API.

Table of Contents

CHAPTER 1. UNDERSTANDING MESSAGE FORMATS **3**
1.1. EXCHANGES 3
1.2. MESSAGES 4
1.3. BUILT-IN TYPE CONVERTERS 8
1.4. BUILT-IN UUID GENERATORS 10

CHAPTER 2. IMPLEMENTING A PROCESSOR **13**
2.1. PROCESSING MODEL 13
2.2. IMPLEMENTING A SIMPLE PROCESSOR 13
2.3. ACCESSING MESSAGE CONTENT 14
2.4. THE EXCHANGEHELPER CLASS 16

CHAPTER 3. TYPE CONVERTERS **18**
3.1. TYPE CONVERTER ARCHITECTURE 18
3.2. IMPLEMENTING TYPE CONVERTER USING ANNOTATIONS 20
3.3. IMPLEMENTING A TYPE CONVERTER DIRECTLY 23

CHAPTER 4. PRODUCER AND CONSUMER TEMPLATES **25**
4.1. USING THE PRODUCER TEMPLATE 25
4.2. USING THE CONSUMER TEMPLATE 39

CHAPTER 5. IMPLEMENTING A COMPONENT **42**
5.1. COMPONENT ARCHITECTURE 42
5.2. HOW TO IMPLEMENT A COMPONENT 49
5.3. AUTO-DISCOVERY AND CONFIGURATION 51

CHAPTER 6. COMPONENT INTERFACE **55**
6.1. THE COMPONENT INTERFACE 55
6.2. IMPLEMENTING THE COMPONENT INTERFACE 56

CHAPTER 7. ENDPOINT INTERFACE **61**
7.1. THE ENDPOINT INTERFACE 61
7.2. IMPLEMENTING THE ENDPOINT INTERFACE 64

CHAPTER 8. CONSUMER INTERFACE **71**
8.1. THE CONSUMER INTERFACE 71
8.2. IMPLEMENTING THE CONSUMER INTERFACE 75

CHAPTER 9. PRODUCER INTERFACE **83**
9.1. THE PRODUCER INTERFACE 83
9.2. IMPLEMENTING THE PRODUCER INTERFACE 85

CHAPTER 10. EXCHANGE INTERFACE **88**
10.1. THE EXCHANGE INTERFACE 88

CHAPTER 11. MESSAGE INTERFACE **92**
11.1. THE MESSAGE INTERFACE 92
11.2. IMPLEMENTING THE MESSAGE INTERFACE 94

INDEX **95**

CHAPTER 1. UNDERSTANDING MESSAGE FORMATS

Abstract

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

1.1. EXCHANGES

Overview

An *exchange object* is a wrapper that encapsulates a received message and stores its associated metadata (including the *exchange properties*). In addition, if the current message is dispatched to a producer endpoint, the exchange provides a temporary slot to hold the reply (the *Out* message).

An important feature of exchanges in Apache Camel is that they support lazy creation of messages. This can provide a significant optimization in the case of routes that do not require explicit access to messages.

Figure 1.1. Exchange Object Passing through a Route

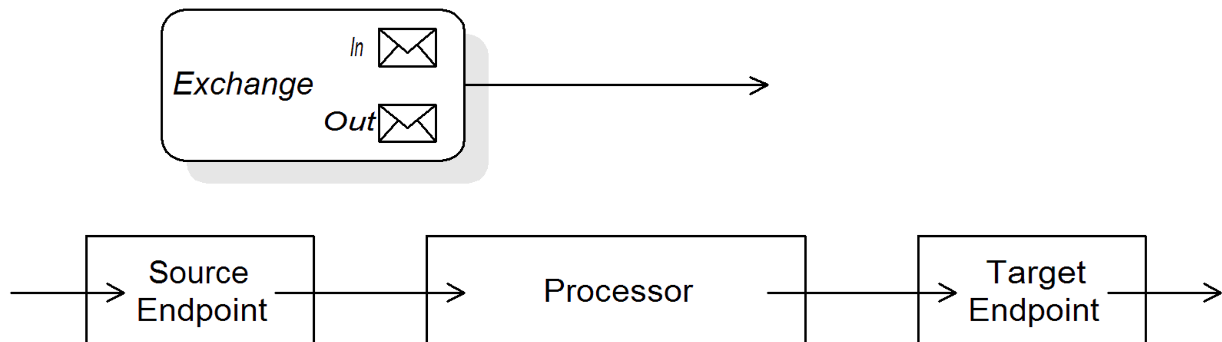


Figure 1.1, "Exchange Object Passing through a Route" shows an exchange object passing through a route. In the context of a route, an exchange object gets passed as the argument of the **`Processor.process()`** method. This means that the exchange object is directly accessible to the source endpoint,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

The Exchange interface

The **`org.apache.camel.Exchange`** interface defines methods to access *In* and *Out* messages, as shown in Example 1.1, "Exchange Methods".

Example 1.1. Exchange Methods

```
// Access the In message
Message getIn();
void    setIn(Message in);

// Access the Out message (if any)
Message getOut();
```

```
void    setOut(Message out);
boolean hasOut();

// Access the exchange ID
String  getExchangeId();
void    setExchangeId(String id);
```

For a complete description of the methods in the **Exchange** interface, see Section 10.1, "The Exchange Interface".

Lazy creation of messages

Apache Camel 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()** or **getOut()**). 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()** or **getOut()**), or if you call an accessor with the boolean argument equal to **true** (that is, **getIn(true)** or **getOut(true)**), the default method implementation creates a new message instance, if one does not already exist.

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

Lazy creation of exchange IDs

Apache Camel supports lazy creation of exchange IDs. You can call **getExchangeId()** on any exchange to obtain a unique ID for that exchange instance, but the ID is generated only when you actually call the method. The **DefaultExchange.getExchangeId()** implementation of this method delegates ID generation to the UUID generator that is registered with the **CamelContext**.

For details of how to register UUID generators with the **CamelContext**, see Section 1.4, "Built-In UUID Generators".

1.2. 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**, which can contain arbitrary MIME types. If you need to obtain a concrete representation of the message contents, you can convert the body and headers to another type using the type converter mechanism and, possibly, using the marshalling and unmarshalling mechanism.

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

The Message interface

The **org.apache.camel.Message** interface defines methods to access the message body, message headers and message attachments, as shown in Example 1.2, “Message Interface”.

Example 1.2. Message Interface

```
// Access the message body
Object getBody();
<T> T  getBody(Class<T> type);
void   setBody(Object body);
<T> void setBody(Object body, Class<T> type);

// Access message headers
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);

// Access message attachments
javax.activation.DataHandler getAttachment(String id);
java.util.Map<String, javax.activation.DataHandler> getAttachments();
java.util.Set<String> getAttachmentNames();
void addAttachment(String id, javax.activation.DataHandler content)

// Access the message ID
String getMessageId();
void   setMessageId(String messageId);
```

For a complete description of the methods in the **Message** interface, see Section 11.1, “The Message Interface”.

Lazy creation of bodies, headers, and attachments

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

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

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

In this route, if we assume that the component referenced by *SourceURL* supports lazy creation, the *In* message headers are not actually parsed until the **header("foo")** call is executed. At that point, the

underlying message implementation parses the headers and populates the header map. The message *body* is not parsed until you reach the end of the route, at the **to("*TargetURL*")** call. At that point, the body is converted into the format required for writing it to the target endpoint, *TargetURL*.

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

Whether or not lazy creation is implemented in practice depends on the underlying component implementation. In general, lazy creation is valuable for those cases where creating a message body, a message header, or a message attachment is expensive. For details about implementing a message type that supports lazy creation, see Section 11.2, “Implementing the Message Interface”.

Lazy creation of message IDs

Apache Camel supports lazy creation of message IDs. That is, a message ID is generated only when you actually call the **getMessageId()** method. The **DefaultExchange.getExchangeId()** implementation of this method delegates ID generation to the UUID generator that is registered with the **CamelContext**.

Some endpoint implementations would call the **getMessageId()** method implicitly, if the endpoint implements a protocol that requires a unique message ID. In particular, JMS messages normally include a header containing unique message ID, so the JMS component automatically calls **getMessageId()** to obtain the message ID (this is controlled by the **messageIdEnabled** option on the JMS endpoint).

For details of how to register UUID generators with the **CamelContext**, see Section 1.4, “Built-In UUID Generators”.

Initial message format

The initial format of an *In* message is determined by the source endpoint, and the initial format of an *Out* message is determined by the target endpoint. If lazy creation is supported by the underlying component, the message remains unparsed until it is accessed explicitly by the application. Most Apache Camel components create the message body in a relatively raw form—for example, representing it using types such as **byte[]**, **ByteBuffer**, **InputStream**, or **OutputStream**. This ensures that the overhead required for creating the initial message is minimal. Where more elaborate message formats are required components usually rely on *type converters* or *marshalling processors*.

Type converters

It does not matter what the initial format of the message is, because you can easily convert a message from one format to another using the built-in type converters (see Section 1.3, “Built-In Type Converters”). There are various methods in the Apache Camel API that expose type conversion functionality. For example, the **convertBodyTo(Class type)** method can be inserted into a route to convert the body of an *In* message, as follows:

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

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

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

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

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

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

Type conversion methods in Message

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

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

For the complete list of supported conversion types, see Section 1.3, “Built-In Type Converters”.

Converting to XML

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

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

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

Marshalling and unmarshalling

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

- **marshal()**
- **unmarshal()**

For example, to read a serialized Java object from a file and unmarshal it into a Java object, you could use the route definition shown in Example 1.3, “Unmarshalling a Java Object”.

Example 1.3. Unmarshalling a Java Object

```
from("file://tmp/appfiles/serialized")
    .unmarshal()
```

```
    .serialization()
    .<FurtherProcessing>
    .to("TargetURL");
```

For details of how to marshal and unmarshal various data formats, see section "Marshalling and unmarshalling" in "Implementing Enterprise Integration Patterns".

Final message format

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

1.3. BUILT-IN TYPE CONVERTERS

Overview

This section describes the conversions supported by the master type converter. These conversions are built into the Apache Camel core.

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

Example 1.4. Converting a Value to a String

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

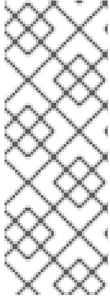
Basic type converters

Apache Camel provides built-in type converters that perform conversions to and from the following basic types:

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

- **`java.io.StringReader`**

However, not all of these types are inter-convertible. The built-in converter is mainly focused on providing conversions from the **`File`** and **`String`** types. The **`File`** type can be converted to any of the preceding types, except **`Reader`**, **`Writer`**, and **`StringReader`**. The **`String`** type can be converted to **`File`**, **`byte[]`**, **`ByteBuffer`**, **`InputStream`**, or **`StringReader`**. The conversion from **`String`** to **`File`** works by interpreting the string as a file name. The trio of **`String`**, **`byte[]`**, and **`ByteBuffer`** are completely inter-convertible.



NOTE

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

Collection type converters

Apache Camel provides built-in type converters that perform conversions to and from the following collection types:

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

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

Map type converters

Apache Camel provides built-in type converters that perform conversions to and from the following map types:

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

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

DOM type converters

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

- **`org.w3c.dom.Document`**—convertible from **`byte[]`**, **`String`**, **`java.io.File`**, and **`java.io.InputStream`**.

- **org.w3c.dom.Node**
- **javax.xml.transform.dom.DOMSource**—convertible from **String**.
- **javax.xml.transform.Source**—convertible from **byte[]** and **String**.

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

SAX type converters

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

- **String**
- **InputStream**
- **Source**
- **StreamSource**
- **DOMSource**

Custom type converters

Apache Camel also enables you to implement your own custom type converters. For details on how to implement a custom type converter, see Chapter 3, *Type Converters*.

1.4. BUILT-IN UUID GENERATORS

Overview

Apache Camel enables you to register a UUID generator in the **CamelContext**. This UUID generator is then used whenever Apache Camel needs to generate a unique ID—in particular, the registered UUID generator is called to generate the IDs returned by the **Exchange.getExchangeId()** and the **Message.getMessageId()** methods.

For example, you might prefer to replace the default UUID generator, if part of your application does not support IDs with a length of 36 characters (like Websphere MQ). Also, it can be convenient to generate IDs using a simple counter (see the **SimpleUuidGenerator**) for testing purposes.

Provided UUID generators

You can configure Apache Camel to use one of the following UUID generators, which are provided in the core:

- **org.apache.camel.impl.ActiveMQUuidGenerator**—*(Default)* generates the same style of ID as is used by Apache ActiveMQ. This implementation might not be suitable for all applications, because it uses some JDK APIs that are forbidden in the context of cloud computing (such as the Google App Engine).

- **org.apache.camel.impl.SimpleUuidGenerator**—implements a simple counter ID, starting at **1**. The underlying implementation uses the **java.util.concurrent.atomic.AtomicLong** type, so that it is thread-safe.
- **org.apache.camel.impl.JavaUuidGenerator**—implements an ID based on the **java.util.UUID** type. Because **java.util.UUID** is synchronized, this might affect performance on some highly concurrent systems.

Custom UUID generator

To implement a custom UUID generator, implement the **org.apache.camel.spi.UuidGenerator** interface, which is shown in Example 1.5, "UuidGenerator Interface". The **generateUuid()** must be implemented to return a unique ID string.

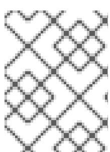
Example 1.5. UuidGenerator Interface

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

/**
 * Generator to generate UUID strings.
 */
public interface UuidGenerator {
    String generateUuid();
}
```

Specifying the UUID generator using Java

To replace the default UUID generator using Java, call the **setUuidGenerator()** method on the current **CamelContext** object. For example, you can register a **SimpleUuidGenerator** instance with the current **CamelContext**, as follows:

```
// Java
getContext().setUuidGenerator(new
org.apache.camel.impl.SimpleUuidGenerator());
```



NOTE

The **setUuidGenerator()** method should be called during startup, *before* any routes are activated.

Specifying the UUID generator using Spring

To replace the default UUID generator using Spring, all you need to do is to create an instance of a UUID generator using the Spring **bean** element. When a **camelContext** instance is created, it automatically looks up the Spring registry, searching for a bean that implements **org.apache.camel.spi.UuidGenerator**. For example, you can register a **SimpleUuidGenerator** instance with the **CamelContext** as follows:

```
<beans ...>
  <bean id="simpleUuidGenerator"
```

```
        class="org.apache.camel.impl.SimpleUuidGenerator" />

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

---

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

CHAPTER 2. IMPLEMENTING A PROCESSOR

Abstract

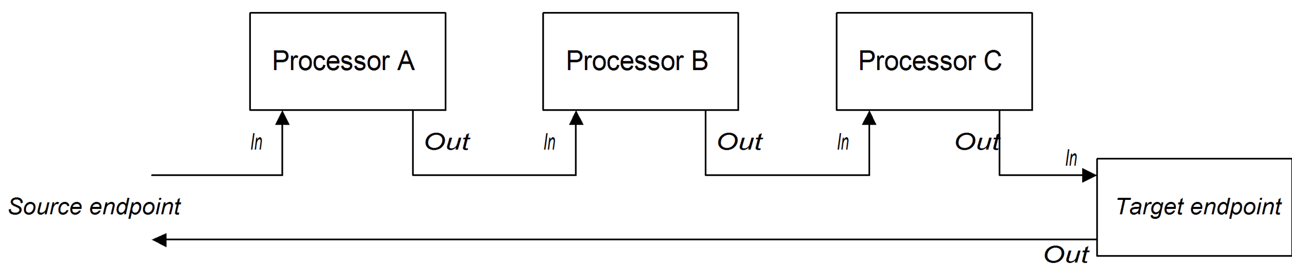
Apache Camel allows you to implement a custom processor. You can then insert the custom processor into a route to perform operations on exchange objects as they pass through the route.

2.1. PROCESSING MODEL

Pipelining model

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

Figure 2.1. Pipelining Model



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

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

Example 2.1. Java DSL Pipeline

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

2.2. IMPLEMENTING A SIMPLE PROCESSOR

Overview

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

Processor interface

Simple processors are created by implementing the **org.apache.camel.Processor** interface. As shown in Example 2.2, "Processor Interface", the interface defines a single method, **process()**, which processes an exchange object.

Example 2.2. Processor Interface

```
package org.apache.camel;

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

Implementing the `Processor` interface

To create a simple processor you must implement the **`Processor`** interface and provide the logic for the **`process()`** method. Example 2.3, “Simple Processor Implementation” shows the outline of a simple processor implementation.

Example 2.3. Simple Processor Implementation

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

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

    public void process(Exchange exchange) throws Exception
    {
        // Insert code that gets executed *before* delegating
        // to the next processor in the chain.
        ...
    }
}
```

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

For examples of how to access the message body and header values inside a simple processor, see Section 2.3, “Accessing Message Content”.

Inserting the simple processor into a route

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

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

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

2.3. ACCESSING MESSAGE CONTENT

Accessing message headers

Message headers typically contain the most useful message content from the perspective of a router,

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

Example 2.4, "Accessing an Authorization Header" shows an example of a custom processor that accesses the value of a header named **Authorization**. This example uses the **ExchangeHelper.getMandatoryHeader()** method, which eliminates the need to test for a null header value.

Example 2.4. Accessing an Authorization Header

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

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

For full details of the **Message** interface, see Section 1.2, "Messages".

Accessing the message body

You can also access the message body. For example, to append a string to the end of the *In* message, you can use the processor shown in Example 2.5, "Accessing the Message Body".

Example 2.5. Accessing the Message Body

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

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

Accessing message attachments

You can access a message's attachments using either the **Message.getAttachment()** method or the **Message.getAttachments()** method. See Example 1.2, "Message Interface" for more details.

2.4. THE EXCHANGEHELPER CLASS

Overview

The **org.apache.camel.util.ExchangeHelper** class is a Apache Camel utility class that provides methods that are useful when implementing a processor.

Resolve an endpoint

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

Example 2.6. The resolveEndpoint() Method

```
public final class ExchangeHelper {
    ...
    @SuppressWarnings({"unchecked" })
    public static Endpoint
    resolveEndpoint(Exchange exchange, Object value)
        throws NoSuchEndpointException { ... }
    ...
}
```

The first argument to **resolveEndpoint()** is an exchange instance, and the second argument is usually an endpoint URI string. Example 2.7, “Creating a File Endpoint” shows how to create a new file endpoint from an exchange instance **exchange**

Example 2.7. Creating a File Endpoint

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

Wrapping the exchange accessors

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

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

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

```
    public static Object getMandatoryInBody(Exchange exchange)
        throws InvalidPayloadException { ... }

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

    public static Object getMandatoryOutBody(Exchange exchange)
        throws InvalidPayloadException { ... }

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

Testing the exchange pattern

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

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

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

Get the In message's MIME content type

If you want to find out the MIME content type of the exchange's *In* message, you can access it by calling the **`ExchangeHelper.getContentType(exchange)`** method. To implement this, the **`ExchangeHelper`** object looks up the value of the *In* message's **`Content-Type`** header—this method relies on the underlying component to populate the header value).

CHAPTER 3. TYPE CONVERTERS

Abstract

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

3.1. TYPE CONVERTER ARCHITECTURE

Overview

This section describes the overall architecture of the type converter mechanism, which you must understand, if you want to write custom type converters. If you only need to use the built-in type converters, see Chapter 1, *Understanding Message Formats*.

Type converter interface

Example 3.1, “TypeConverter Interface” shows the definition of the **`org.apache.camel.TypeConverter`** interface, which all type converters must implement.

Example 3.1. TypeConverter Interface

```
package org.apache.camel;

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

Master type converter

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

For users of the type conversion mechanism, the master type converter is the most important because it provides the entry point for accessing the conversion mechanism. During start up, Apache Camel automatically associates a master type converter instance with the **`CamelContext`** object. To obtain a reference to the master type converter, you call the **`CamelContext.getTypeConverter()`** method. For example, if you have an exchange object, **`exchange`**, you can obtain a reference to the master type converter as shown in Example 3.2, “Getting a Master Type Converter”.

Example 3.2. Getting a Master Type Converter

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

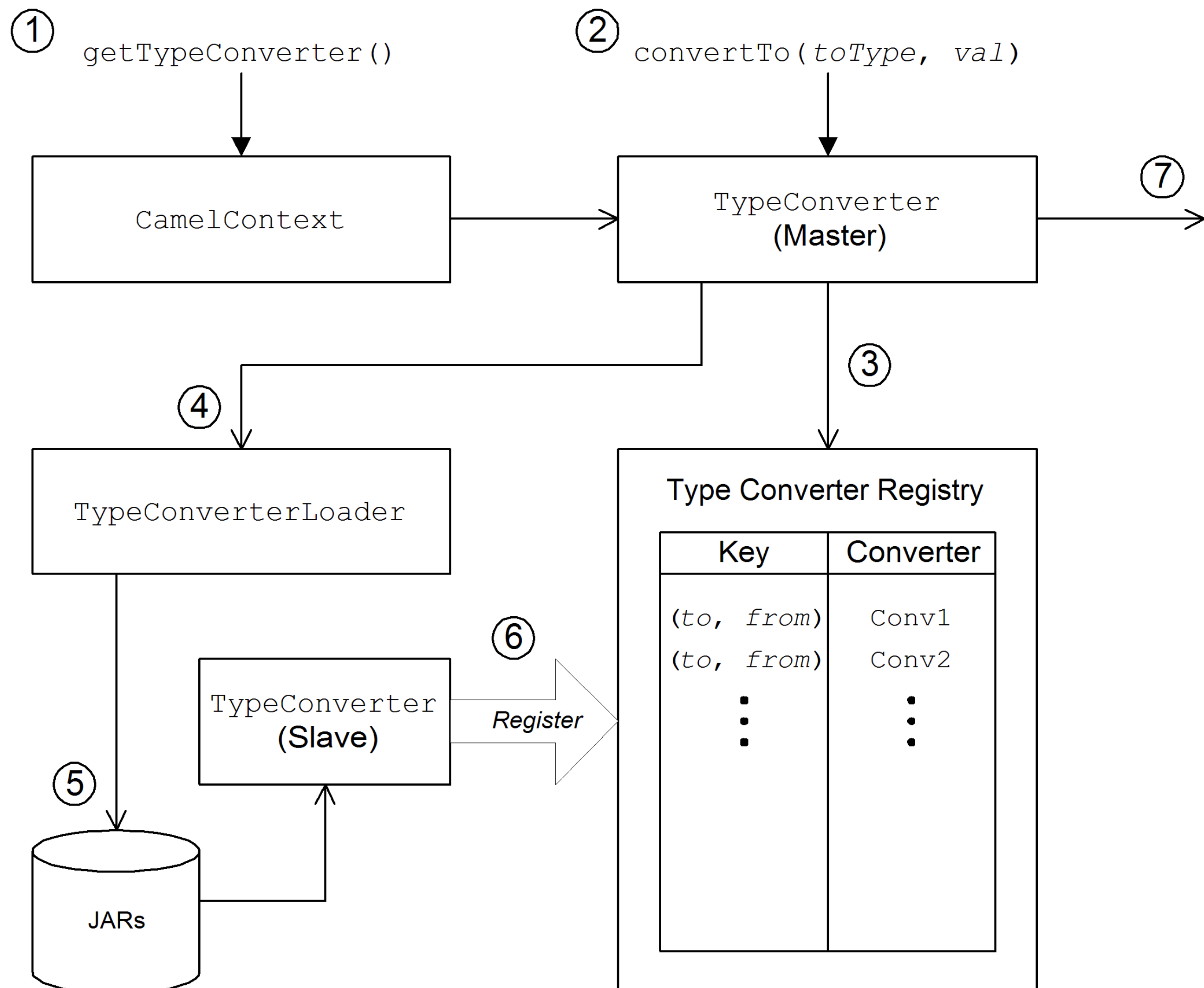
Type converter loader

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

Type conversion process

Figure 3.1, "Type Conversion Process" gives an overview of the type conversion process, showing the steps involved in converting a given data value, **value**, to a specified type, **toType**.

Figure 3.1. Type Conversion Process



The type conversion mechanism proceeds as follows:

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

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

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

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

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

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

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

3.2. IMPLEMENTING TYPE CONVERTER USING ANNOTATIONS

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 Apache Camel, so that it is automatically loaded by the annotation type converter loader.

How to implement a type converter

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

1. Implement an annotated converter class.

2. Create a TypeConverter file.

3. Package the type converter.

Implement an annotated converter class

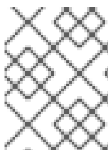
You can implement a custom type converter class using the **@Converter** annotation. You must annotate the class itself and each of the **static** methods intended to perform type conversion. Each converter method takes an argument that defines the *from* type, optionally takes a second **Exchange** argument, and has a non-void return value that defines the *to* type. The type converter loader uses Java reflection to find the annotated methods and integrate them into the type converter mechanism. Example 3.3, “Example of an Annotated Converter Class” shows an example of an annotated converter class that defines a converter method for converting from **java.io.File** to **java.io.InputStream** and another converter method (with an **Exchange** argument) for converting from **byte[]** to **String**.

Example 3.3. Example of an Annotated Converter Class

```
package com.YourDomain.YourPackageName;
```

```
import org.apache.camel.Converter;

import java.io.*;

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

    @Converter
    public static InputStream toInputStream(File file) throws
FileNotFoundException {
        return new BufferedInputStream(new FileInputStream(file));
    }

    @Converter
    public static String toString(byte[] data, Exchange exchange) {
        if (exchange != null) {
            String charsetName =
exchange.getProperty(Exchange.CHARSET_NAME, String.class);
            if (charsetName != null) {
                try {
                    return new String(data, charsetName);
                } catch (UnsupportedEncodingException e) {
                    LOG.warn("Can't convert the byte to String with the
charset " + charsetName, e);
                }
            }
        }
        return new String(data);
    }
}
```

The **toInputStream()** method is responsible for performing the conversion from the **File** type to the **InputStream** type and the **toString()** method is responsible for performing the conversion from the **byte[]** type to the **String** type.



NOTE

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

Create a TypeConverter file

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

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

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

TypeConverter file would have the following contents:

```
com.YourDomain.YourPackageName
```

Package the type converter

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

Fallback converter method

In addition to defining regular converter methods using the **@Converter** annotation, you can optionally define a fallback converter method using the **@FallbackConverter** annotation. The fallback converter method will only be tried, if the master type converter fails to find a regular converter method in the type registry.

The essential difference between a regular converter method and a fallback converter method is that whereas a regular converter is defined to perform conversion between a specific pair of types (for example, from **byte[]** to **String**), a fallback converter can potentially perform conversion between *any* pair of types. It is up to the code in the body of the fallback converter method to figure out which conversions it is able to perform. At run time, if a conversion cannot be performed by a regular converter, the master type converter iterates through every available fallback converter until it finds one that can perform the conversion.

The method signature of a fallback converter can have either of the following forms:

```
// 1. Non-generic form of signature
@FallbackConverter
public static Object MethodName(
    Class type,
    Exchange exchange,
    Object value,
    TypeConverterRegistry registry
)

// 2. Templating form of signature
@FallbackConverter
public static <T> T MethodName(
    Class<T> type,
    Exchange exchange,
    Object value,
    TypeConverterRegistry registry
)
```

Where *MethodName* is an arbitrary method name for the fallback converter.

For example, the following code extract (taken from the implementation of the File component) shows a fallback converter that can convert the body of a **GenericFile** object, exploiting the type converters already available in the type converter registry:

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

import org.apache.camel.Converter;
```

```
import org.apache.camel.FallbackConverter;
import org.apache.camel.Exchange;
import org.apache.camel.TypeConverter;
import org.apache.camel.spi.TypeConverterRegistry;

@Converter
public final class GenericFileConverter {

    private GenericFileConverter() {
        // Helper Class
    }

    @FallbackConverter
    public static <T> T convertTo(Class<T> type, Exchange exchange, Object
value, TypeConverterRegistry registry) {
        // use a fallback type converter so we can convert the embedded
body if the value is GenericFile
        if (GenericFile.class.isAssignableFrom(value.getClass())) {
            GenericFile file = (GenericFile) value;
            Class from = file.getBody().getClass();
            TypeConverter tc = registry.lookup(type, from);
            if (tc != null) {
                Object body = file.getBody();
                return tc.convertTo(type, exchange, body);
            }
        }

        return null;
    }
    ...
}
```

3.3. IMPLEMENTING A TYPE CONVERTER DIRECTLY

Overview

Generally, the recommended way to implement a type converter is to use an annotated class, as described in the previous section, Section 3.2, “Implementing Type Converter Using Annotations”. But if you want to have complete control over the registration of your type converter, you can implement a custom slave type converter and add it directly to the type converter registry, as described here.

Implement the TypeConverter interface

To implement your own type converter class, define a class that implements the **TypeConverter** interface. For example, the following **MyOrderTypeConverter** class converts an integer value to a **MyOrder** object, where the integer value is used to initialize the order ID in the **MyOrder** object.

```
import org.apache.camel.TypeConverter

private class MyOrderTypeConverter implements TypeConverter {

    public <T> T convertTo(Class<T> type, Object value) {
        // converter from value to the MyOrder bean
        MyOrder order = new MyOrder();
```

```
        order.setId(Integer.parseInt(value.toString()));
        return (T) order;
    }

    public <T> T convertTo(Class<T> type, Exchange exchange, Object value)
{
        // this method with the Exchange parameter will be preferd by
Camel to invoke
        // this allows you to fetch information from the exchange during
convertions
        // such as an encoding parameter or the likes
        return convertTo(type, value);
    }

    public <T> T mandatoryConvertTo(Class<T> type, Object value) {
        return convertTo(type, value);
    }

    public <T> T mandatoryConvertTo(Class<T> type, Exchange exchange,
Object value) {
        return convertTo(type, value);
    }
}
```

Add the type converter to the registry

You can add the custom type converter *directly* to the type converter registry using code like the following:

```
// Add the custom type converter to the type converter registry
context.getTypeConverterRegistry().addTypeConverter(MyOrder.class,
String.class, new MyOrderTypeConverter());
```

Where **`context`** is the current **`org.apache.camel.CamelContext`** instance. The **`addTypeConverter()`** method registers the **`MyOrderTypeConverter`** class against the specific type conversion, from **`String.class`** to **`MyOrder.class`**.

CHAPTER 4. PRODUCER AND CONSUMER TEMPLATES

Abstract

The producer and consumer templates in Apache Camel are modelled after a feature of the Spring container API, whereby access to a resource is provided through a simplified, easy-to-use API known as a *template*. In the case of Apache Camel, the producer template and consumer template provide simplified interfaces for sending messages to and receiving messages from producer endpoints and consumer endpoints.

4.1. USING THE PRODUCER TEMPLATE

4.1.1. Introduction to the Producer Template

Overview

The producer template supports a variety of different approaches to invoking producer endpoints. There are methods that support different formats for the request message (as an **Exchange** object, as a message body, as a message body with a single header setting, and so on) and there are methods to support both the synchronous and the asynchronous style of invocation. Overall, producer template methods can be grouped into the following categories:

- the section called “Synchronous invocation”.
- the section called “Synchronous invocation with a processor”.
- the section called “Asynchronous invocation”.
- the section called “Asynchronous invocation with a callback”.

Synchronous invocation

The methods for invoking endpoints synchronously have names of the form **send*Suffix*()** and **request*Suffix*()**. For example, the methods for invoking an endpoint using either the default message exchange pattern (MEP) or an explicitly specified MEP are named **send()**, **sendBody()**, and **sendBodyAndHeader()** (where these methods respectively send an **Exchange** object, a message body, or a message body and header value). If you want to force the MEP to be *InOut* (request/reply semantics), you can call the **request()**, **requestBody()**, and **requestBodyAndHeader()** methods instead.

The following example shows how to create a **ProducerTemplate** instance and use it to send a message body to the **activemq:MyQueue** endpoint. The example also shows how to send a message body and header value using **sendBodyAndHeader()**.

```
import org.apache.camel.ProducerTemplate
import org.apache.camel.impl.DefaultProducerTemplate
...
ProducerTemplate template = context.createProducerTemplate();

// Send to a specific queue
template.sendBody("activemq:MyQueue", "<hello>world!</hello>");
```

```
// Send with a body and header
template.sendBodyAndHeader(
    "activemq:MyQueue",
    "<hello>world!</hello>",
    "CustomerRating", "Gold" );
```

Synchronous invocation with a processor

A special case of synchronous invocation is where you provide the **send()** method with a **Processor** argument instead of an **Exchange** argument. In this case, the producer template implicitly asks the specified endpoint to create an **Exchange** instance (typically, but not always having the *InOnly* MEP by default). This default exchange is then passed to the processor, which initializes the contents of the exchange object.

The following example shows how to send an exchange initialized by the **MyProcessor** processor to the **activemq:MyQueue** endpoint.

```
import org.apache.camel.ProducerTemplate
import org.apache.camel.impl.DefaultProducerTemplate
...
ProducerTemplate template = context.createProducerTemplate();

// Send to a specific queue, using a processor to initialize
template.send("activemq:MyQueue", new MyProcessor());
```

The **MyProcessor** class is implemented as shown in the following example. In addition to setting the *In* message body (as shown here), you could also initialize message heades and exchange properties.

```
import org.apache.camel.Processor;
import org.apache.camel.Exchange;
...
public class MyProcessor implements Processor {
    public MyProcessor() { }

    public void process(Exchange ex) {
        ex.getIn().setBody("<hello>world!</hello>");
    }
}
```

Asynchronous invocation

The methods for invoking endpoints *asynchronously* have names of the form **asyncSend*Suffix*()** and **asyncRequest*Suffix*()**. For example, the methods for invoking an endpoint using either the default message exchange pattern (MEP) or an explicitly specified MEP are named **asyncSend()** and **asyncSendBody()** (where these methods respectively send an **Exchange** object or a message body). If you want to force the MEP to be *InOut* (request/reply semantics), you can call the **asyncRequestBody()**, **asyncRequestBodyAndHeader()**, and **asyncRequestBodyAndHeaders()** methods instead.

The following example shows how to send an exchange asynchronously to the **direct:start** endpoint. The **asyncSend()** method returns a **java.util.concurrent.Future** object, which is used to retrieve the invocation result at a later time.

```
import java.util.concurrent.Future;

import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultExchange;
...
Exchange exchange = new DefaultExchange(context);
exchange.getIn().setBody("Hello");

Future<Exchange> future = template.asyncSend("direct:start", exchange);

// You can do other things, whilst waiting for the invocation to complete
...
// Now, retrieve the resulting exchange from the Future
Exchange result = future.get();
```

The producer template also provides methods to send a message body asynchronously (for example, using **asyncSendBody()** or **asyncRequestBody()**). In this case, you can use one of the following helper methods to extract the returned message body from the **Future** object:

```
<T> T extractFutureBody(Future future, Class<T> type);
<T> T extractFutureBody(Future future, long timeout, TimeUnit unit,
Class<T> type) throws TimeoutException;
```

The first version of the **extractFutureBody()** method blocks until the invocation completes and the reply message is available. The second version of the **extractFutureBody()** method allows you to specify a timeout. Both methods have a type argument, **type**, which casts the returned message body to the specified type using a built-in type converter.

The following example shows how to use the **asyncRequestBody()** method to send a message body to the **direct:start** endpoint. The blocking **extractFutureBody()** method is then used to retrieve the reply message body from the **Future** object.

```
Future<Object> future = template.asyncRequestBody("direct:start",
"Hello");

// You can do other things, whilst waiting for the invocation to complete
...
// Now, retrieve the reply message body as a String type
String result = template.extractFutureBody(future, String.class);
```

Asynchronous invocation with a callback

In the preceding asynchronous examples, the request message is dispatched in a sub-thread, while the reply is retrieved and processed by the main thread. The producer template also gives you the option, however, of processing replies in the sub-thread, using one of the **asyncCallback()**, **asyncCallbackSendBody()**, or **asyncCallbackRequestBody()** methods. In this case, you supply a callback object (of **org.apache.camel.impl.SynchronizationAdapter** type), which automatically gets invoked in the sub-thread as soon as a reply message arrives.

The **Synchronization** callback interface is defined as follows:

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

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

public interface Synchronization {
    void onComplete(Exchange exchange);
    void onFailure(Exchange exchange);
}
```

Where the **onComplete()** method is called on receipt of a normal reply and the **onFailure()** method is called on receipt of a fault message reply. Only one of these methods gets called back, so you must override both of them to ensure that all types of reply are processed.

The following example shows how to send an exchange to the **direct:start** endpoint, where the reply message is processed in the sub-thread by the **SynchronizationAdapter** callback object.

```
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

import org.apache.camel.Exchange;
import org.apache.camel.impl.DefaultExchange;
import org.apache.camel.impl.SynchronizationAdapter;
...
Exchange exchange = context.getEndpoint("direct:start").createExchange();
exchange.getIn().setBody("Hello");

Future<Exchange> future = template.asyncCallback("direct:start", exchange,
new SynchronizationAdapter() {
    @Override
    public void onComplete(Exchange exchange) {
        assertEquals("Hello World", exchange.getIn().getBody());
    }
});
```

Where the **SynchronizationAdapter** class is a default implementation of the **Synchronization** interface, which you can override to provide your own definitions of the **onComplete()** and **onFailure()** callback methods.

You still have the option of accessing the reply from the main thread, because the **asyncCallback()** method also returns a **Future** object—for example:

```
// Retrieve the reply from the main thread, specifying a timeout
Exchange reply = future.get(10, TimeUnit.SECONDS);
```

4.1.2. Synchronous Send

Overview

The *synchronous send* methods are a collection of methods that you can use to invoke a producer endpoint, where the current thread blocks until the method invocation is complete and the reply (if any) has been received. These methods are compatible with any kind of message exchange protocol.

Send an exchange

The basic **send()** method is a general-purpose method that sends the contents of an **Exchange** object

to an endpoint, using the message exchange pattern (MEP) of the exchange. The return value is the exchange that you get after it has been processed by the producer endpoint (possibly containing an *Out* message, depending on the MEP).

There are three varieties of **send()** method for sending an exchange that let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object.

```
Exchange send(Exchange exchange);
Exchange send(String endpointUri, Exchange exchange);
Exchange send(Endpoint endpoint, Exchange exchange);
```

Send an exchange populated by a processor

A simple variation of the general **send()** method is to use a processor to populate a default exchange, instead of supplying the exchange object explicitly (see the section called “Synchronous invocation with a processor” for details).

The **send()** methods for sending an exchange populated by a processor let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object. In addition, you can optionally specify the exchange's MEP by supplying the **pattern** argument, instead of accepting the default.

```
Exchange send(Processor processor);
Exchange send(String endpointUri, Processor processor);
Exchange send(Endpoint endpoint, Processor processor);
Exchange send(
    String endpointUri,
    ExchangePattern pattern,
    Processor processor
);
Exchange send(
    Endpoint endpoint,
    ExchangePattern pattern,
    Processor processor
);
```

Send a message body

If you are only concerned with the contents of the message body that you want to send, you can use the **sendBody()** methods to provide the message body as an argument and let the producer template take care of inserting the body into a default exchange object.

The **sendBody()** methods let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object. In addition, you can optionally specify the exchange's MEP by supplying the **pattern** argument, instead of accepting the default. The methods *without* a **pattern** argument return **void** (even though the invocation might give rise to a reply in some cases); and the methods *with* a **pattern** argument return either the body of the *Out* message (if there is one) or the body of the *In* message (otherwise).

```
void sendBody(Object body);
void sendBody(String endpointUri, Object body);
void sendBody(Endpoint endpoint, Object body);
Object sendBody(
```

```
    String endpointUri,
    ExchangePattern pattern,
    Object body
);
Object sendBody(
    Endpoint endpoint,
    ExchangePattern pattern,
    Object body
);
```

Send a message body and header(s)

For testing purposes, it is often interesting to try out the effect of a *single* header setting and the **sendBodyAndHeader()** methods are useful for this kind of header testing. You supply the message body and header setting as arguments to **sendBodyAndHeader()** and let the producer template take care of inserting the body and header setting into a default exchange object.

The **sendBodyAndHeader()** methods let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object. In addition, you can optionally specify the exchange's MEP by supplying the **pattern** argument, instead of accepting the default. The methods *without* a **pattern** argument return **void** (even though the invocation might give rise to a reply in some cases); and the methods *with* a **pattern** argument return either the body of the *Out* message (if there is one) or the body of the *In* message (otherwise).

```
void sendBodyAndHeader(
    Object body,
    String header,
    Object headerValue
);
void sendBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue
);
void sendBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue
);
Object sendBodyAndHeader(
    String endpointUri,
    ExchangePattern pattern,
    Object body,
    String header,
    Object headerValue
);
Object sendBodyAndHeader(
    Endpoint endpoint,
    ExchangePattern pattern,
    Object body,
    String header,
    Object headerValue
);
```

The **sendBodyAndHeaders()** methods are similar to the **sendBodyAndHeader()** methods, except that instead of supplying just a single header setting, these methods allow you to specify a complete hash map of header settings.

```
void sendBodyAndHeaders(
    Object body,
    Map<String, Object> headers
);
void sendBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers
);
void sendBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers
);
Object sendBodyAndHeaders(
    String endpointUri,
    ExchangePattern pattern,
    Object body,
    Map<String, Object> headers
);
Object sendBodyAndHeaders(
    Endpoint endpoint,
    ExchangePattern pattern,
    Object body,
    Map<String, Object> headers
);
```

Send a message body and exchange property

You can try out the effect of setting a single exchange property using the **sendBodyAndProperty()** methods. You supply the message body and property setting as arguments to **sendBodyAndProperty()** and let the producer template take care of inserting the body and exchange property into a default exchange object.

The **sendBodyAndProperty()** methods let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **Endpoint** object. In addition, you can optionally specify the exchange's MEP by supplying the **pattern** argument, instead of accepting the default. The methods *without* a **pattern** argument return **void** (even though the invocation might give rise to a reply in some cases); and the methods *with* a **pattern** argument return either the body of the *Out* message (if there is one) or the body of the *In* message (otherwise).

```
void sendBodyAndProperty(
    Object body,
    String property,
    Object propertyValue
);
void sendBodyAndProperty(
    String endpointUri,
    Object body,
    String property,
```

```
    Object propertyValue
);
void sendBodyAndProperty(
    Endpoint endpoint,
    Object body,
    String property,
    Object propertyValue
);
Object sendBodyAndProperty(
    String endpoint,
    ExchangePattern pattern,
    Object body,
    String property,
    Object propertyValue
);
Object sendBodyAndProperty(
    Endpoint endpoint,
    ExchangePattern pattern,
    Object body,
    String property,
    Object propertyValue
);
```

4.1.3. Synchronous Request with InOut Pattern

Overview

The *synchronous request* methods are similar to the synchronous send methods, except that the request methods force the message exchange pattern to be *InOut* (conforming to request/reply semantics). Hence, it is generally convenient to use a synchronous request method, if you expect to receive a reply from the producer endpoint.

Request an exchange populated by a processor

The basic **`request()`** method is a general-purpose method that uses a processor to populate a default exchange and forces the message exchange pattern to be *InOut* (so that the invocation obeys request/reply semantics). The return value is the exchange that you get after it has been processed by the producer endpoint, where the *Out* message contains the reply message.

The **`request()`** methods for sending an exchange populated by a processor let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **`Endpoint`** object.

```
Exchange request(String endpointUri, Processor processor);
Exchange request(Endpoint endpoint, Processor processor);
```

Request a message body

If you are only concerned with the contents of the message body in the request and in the reply, you can use the **`requestBody()`** methods to provide the request message body as an argument and let the producer template take care of inserting the body into a default exchange object.

The **`requestBody()`** methods let you specify the target endpoint in one of the following ways: as the default endpoint, as an endpoint URI, or as an **`Endpoint`** object. The return value is the body of the reply message (*Out* message body), which can either be returned as plain **`Object`** or converted to a

specific type, **T**, using the built-in type converters (see Section 1.3, “Built-In Type Converters”).

```
Object requestBody(Object body);
<T> T requestBody(Object body, Class<T> type);
Object requestBody(
    String endpointUri,
    Object body
);
<T> T requestBody(
    String endpointUri,
    Object body,
    Class<T> type
);
Object requestBody(
    Endpoint endpoint,
    Object body
);
<T> T requestBody(
    Endpoint endpoint,
    Object body,
    Class<T> type
);
```

Request a message body and header(s)

You can try out the effect of setting a single header value using the **requestBodyAndHeader()** methods. You supply the message body and header setting as arguments to **requestBodyAndHeader()** and let the producer template take care of inserting the body and exchange property into a default exchange object.

The **requestBodyAndHeader()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object. The return value is the body of the reply message (*Out* message body), which can either be returned as plain **Object** or converted to a specific type, **T**, using the built-in type converters (see Section 1.3, “Built-In Type Converters”).

```
Object requestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue
);
<T> T requestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
Object requestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue
);
<T> T requestBodyAndHeader(
```

```
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
```

The **requestBodyAndHeaders()** methods are similar to the **requestBodyAndHeader()** methods, except that instead of supplying just a single header setting, these methods allow you to specify a complete hash map of header settings.

```
Object requestBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers
);
<T> T requestBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers,
    Class<T> type
);
Object requestBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers
);
<T> T requestBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers,
    Class<T> type
);
```

4.1.4. Asynchronous Send

Overview

The producer template provides a variety of methods for invoking a producer endpoint asynchronously, so that the main thread does not block while waiting for the invocation to complete and the reply message can be retrieved at a later time. The asynchronous send methods described in this section are compatible with any kind of message exchange protocol.

Send an exchange

The basic **asyncSend()** method takes an **Exchange** argument and invokes an endpoint asynchronously, using the message exchange pattern (MEP) of the specified exchange. The return value is a **java.util.concurrent.Future** object, which is a ticket you can use to collect the reply message at a later time—for details of how to obtain the return value from the **Future** object, see the section called “Asynchronous invocation”.

The following **asyncSend()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Exchange> asyncSend(String endpointUri, Exchange exchange);
Future<Exchange> asyncSend(Endpoint endpoint, Exchange exchange);
```

Send an exchange populated by a processor

A simple variation of the general **asyncSend()** method is to use a processor to populate a default exchange, instead of supplying the exchange object explicitly.

The following **asyncSend()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Exchange> asyncSend(String endpointUri, Processor processor);
Future<Exchange> asyncSend(Endpoint endpoint, Processor processor);
```

Send a message body

If you are only concerned with the contents of the message body that you want to send, you can use the **asyncSendBody()** methods to send a message body asynchronously and let the producer template take care of inserting the body into a default exchange object.

The **asyncSendBody()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Object> asyncSendBody(String endpointUri, Object body);
Future<Object> asyncSendBody(Endpoint endpoint, Object body);
```

4.1.5. Asynchronous Request with InOut Pattern

Overview

The *asynchronous request* methods are similar to the asynchronous send methods, except that the request methods force the message exchange pattern to be *InOut* (conforming to request/reply semantics). Hence, it is generally convenient to use an asynchronous request method, if you expect to receive a reply from the producer endpoint.

Request a message body

If you are only concerned with the contents of the message body in the request and in the reply, you can use the **requestBody()** methods to provide the request message body as an argument and let the producer template take care of inserting the body into a default exchange object.

The **asyncRequestBody()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object. The return value that is retrievable from the **Future** object is the body of the reply message (*Out* message body), which can be returned either as a plain **Object** or converted to a specific type, **T**, using a built-in type converter (see the section called “Asynchronous invocation”).

```
Future<Object> asyncRequestBody(
    String endpointUri,
    Object body
);
<T> Future<T> asyncRequestBody(
```

```
    String endpointUri,
    Object body,
    Class<T> type
);
Future<Object> asyncRequestBody(
    Endpoint endpoint,
    Object body
);
<T> Future<T> asyncRequestBody(
    Endpoint endpoint,
    Object body,
    Class<T> type
);
```

Request a message body and header(s)

You can try out the effect of setting a single header value using the **asyncRequestBodyAndHeader()** methods. You supply the message body and header setting as arguments to **asyncRequestBodyAndHeader()** and let the producer template take care of inserting the body and exchange property into a default exchange object.

The **asyncRequestBodyAndHeader()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object. The return value that is retrievable from the **Future** object is the body of the reply message (*Out* message body), which can be returned either as a plain **Object** or converted to a specific type, **T**, using a built-in type converter (see the section called “Asynchronous invocation”).

```
Future<Object> asyncRequestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue
);
<T> Future<T> asyncRequestBodyAndHeader(
    String endpointUri,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
Future<Object> asyncRequestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue
);
<T> Future<T> asyncRequestBodyAndHeader(
    Endpoint endpoint,
    Object body,
    String header,
    Object headerValue,
    Class<T> type
);
```

The **asyncRequestBodyAndHeaders()** methods are similar to the **asyncRequestBodyAndHeader()** methods, except that instead of supplying just a single header setting, these methods allow you to specify a complete hash map of header settings.

```
Future<Object> asyncRequestBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers
);
<T> Future<T> asyncRequestBodyAndHeaders(
    String endpointUri,
    Object body,
    Map<String, Object> headers,
    Class<T> type
);
Future<Object> asyncRequestBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers
);
<T> Future<T> asyncRequestBodyAndHeaders(
    Endpoint endpoint,
    Object body,
    Map<String, Object> headers,
    Class<T> type
);
```

4.1.6. Asynchronous Send with Callback

Overview

The producer template also provides the option of processing the reply message in the same sub-thread that is used to invoke the producer endpoint. In this case, you provide a callback object, which automatically gets invoked in the sub-thread as soon as the reply message is received. In other words, the *asynchronous send with callback* methods enable you to initiate an invocation in your main thread and then have all of the associated processing—invocation of the producer endpoint, waiting for a reply and processing the reply—occur asynchronously in a sub-thread.

Send an exchange

The basic **asyncCallback()** method takes an **Exchange** argument and invokes an endpoint asynchronously, using the message exchange pattern (MEP) of the specified exchange. This method is similar to the **asyncSend()** method for exchanges, except that it takes an additional **org.apache.camel.spi.Synchronization** argument, which is a callback interface with two methods: **onComplete()** and **onFailure()**. For details of how to use the **Synchronization** callback, see the section called “Asynchronous invocation with a callback”.

The following **asyncCallback()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Exchange> asyncCallback(
    String endpointUri,
    Exchange exchange,
    Synchronization onCompletion
```

```
);
Future<Exchange> asyncCallback(
    Endpoint endpoint,
    Exchange exchange,
    Synchronization onCompletion
);
```

Send an exchange populated by a processor

The **asyncCallback()** method for processors calls a processor to populate a default exchange and forces the message exchange pattern to be *InOut* (so that the invocation obeys request/reply semantics).

The following **asyncCallback()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Exchange> asyncCallback(
    String endpointUri,
    Processor processor,
    Synchronization onCompletion
);
Future<Exchange> asyncCallback(
    Endpoint endpoint,
    Processor processor,
    Synchronization onCompletion
);
```

Send a message body

If you are only concerned with the contents of the message body that you want to send, you can use the **asyncCallbackSendBody()** methods to send a message body asynchronously and let the producer template take care of inserting the body into a default exchange object.

The **asyncCallbackSendBody()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Object> asyncCallbackSendBody(
    String endpointUri,
    Object body,
    Synchronization onCompletion
);
Future<Object> asyncCallbackSendBody(
    Endpoint endpoint,
    Object body,
    Synchronization onCompletion
);
```

Request a message body

If you are only concerned with the contents of the message body in the request and in the reply, you can use the **asyncCallbackRequestBody()** methods to provide the request message body as an argument and let the producer template take care of inserting the body into a default exchange object.

The **asyncCallbackRequestBody()** methods let you specify the target endpoint in one of the following ways: as an endpoint URI, or as an **Endpoint** object.

```
Future<Object> asyncCallbackRequestBody(
    String endpointUri,
    Object body,
    Synchronization onCompletion
);
Future<Object> asyncCallbackRequestBody(
    Endpoint endpoint,
    Object body,
    Synchronization onCompletion
);
```

4.2. USING THE CONSUMER TEMPLATE

Overview

The consumer template provides methods for polling a consumer endpoint in order to receive incoming messages. You can choose to receive the incoming message either in the form of an exchange object or in the form of a message body (where the message body can be cast to a particular type using a built-in type converter).

Example of polling exchanges

You can use a consumer template to poll a consumer endpoint for exchanges using one of the following polling methods: blocking **receive()**; **receive()** with a timeout; or **receiveNoWait()**, which returns immediately. Because a consumer endpoint represents a service, it is also essential to start the service thread by calling **start()** before you attempt to poll for exchanges.

The following example shows how to poll an exchange from the **seda:foo** consumer endpoint using the blocking **receive()** method:

```
import org.apache.camel.ProducerTemplate;
import org.apache.camel.ConsumerTemplate;
import org.apache.camel.Exchange;
...
ProducerTemplate template = context.createProducerTemplate();
ConsumerTemplate consumer = context.createConsumerTemplate();

// Start the consumer service
consumer.start();
...
template.sendBody("seda:foo", "Hello");
Exchange out = consumer.receive("seda:foo");
...
// Stop the consumer service
consumer.stop();
```

Where the consumer template instance, **consumer**, is instantiated using the **CamelContext.createConsumerTemplate()** method and the consumer service thread is started by calling **ConsumerTemplate.start()**.

Example of polling message bodies

You can also poll a consumer endpoint for incoming message bodies using one of the following methods: blocking **receiveBody()**; **receiveBody()** with a timeout; or **receiveBodyNoWait()**, which returns immediately. As in the previous example, it is also essential to start the service thread by calling **start()** before you attempt to poll for exchanges.

The following example shows how to poll an incoming message body from the **seda:foo** consumer endpoint using the blocking **receiveBody()** method:

```
import org.apache.camel.ProducerTemplate;
import org.apache.camel.ConsumerTemplate;
...
ProducerTemplate template = context.createProducerTemplate();
ConsumerTemplate consumer = context.createConsumerTemplate();

// Start the consumer service
consumer.start();
...
template.sendBody("seda:foo", "Hello");
Object body = consumer.receiveBody("seda:foo");
...
// Stop the consumer service
consumer.stop();
```

Methods for polling exchanges

There are three basic methods for polling *exchanges* from a consumer endpoint: **receive()** without a timeout blocks indefinitely; **receive()** with a timeout blocks for the specified period of milliseconds; and **receiveNoWait()** is non-blocking. You can specify the consumer endpoint either as an endpoint URI or as an **Endpoint** instance.

```
Exchange receive(String endpointUri);
Exchange receive(String endpointUri, long timeout);
Exchange receiveNoWait(String endpointUri);

Exchange receive(Endpoint endpoint);
Exchange receive(Endpoint endpoint, long timeout);
Exchange receiveNoWait(Endpoint endpoint);
```

Methods for polling message bodies

There are three basic methods for polling *message bodies* from a consumer endpoint: **receiveBody()** without a timeout blocks indefinitely; **receiveBody()** with a timeout blocks for the specified period of milliseconds; and **receiveBodyNoWait()** is non-blocking. You can specify the consumer endpoint either as an endpoint URI or as an **Endpoint** instance. Moreover, by calling the templating forms of these methods, you can convert the returned body to a particular type, **T**, using a built-in type converter.

```
Object receiveBody(String endpointUri);
Object receiveBody(String endpointUri, long timeout);
Object receiveBodyNoWait(String endpointUri);

Object receiveBody(Endpoint endpoint);
```

```
Object receiveBody(Endpoint endpoint, long timeout);
Object receiveBodyNoWait(Endpoint endpoint);

<T> T receiveBody(String endpointUri, Class<T> type);
<T> T receiveBody(String endpointUri, long timeout, Class<T> type);
<T> T receiveBodyNoWait(String endpointUri, Class<T> type);

<T> T receiveBody(Endpoint endpoint, Class<T> type);
<T> T receiveBody(Endpoint endpoint, long timeout, Class<T> type);
<T> T receiveBodyNoWait(Endpoint endpoint, Class<T> type);
```

CHAPTER 5. IMPLEMENTING A COMPONENT

Abstract

This chapter provides a general overview of the approaches can be used to implement a Apache Camel component.

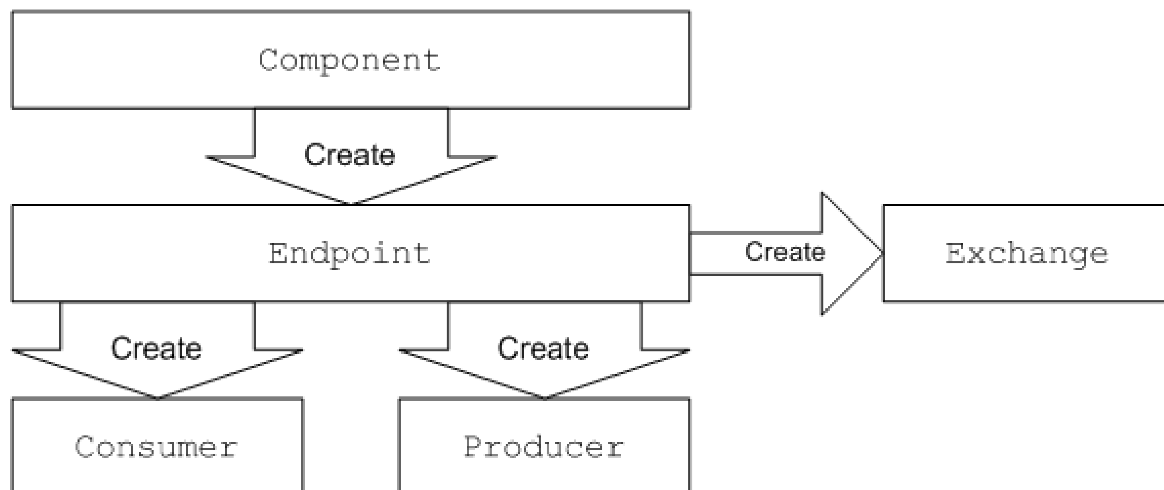
5.1. COMPONENT ARCHITECTURE

5.1.1. Factory Patterns for a Component

Overview

A Apache Camel component consists of a set of classes that are related to each other through a factory pattern. The primary entry point to a component is the **`Component`** object itself (an instance of **`org.apache.camel.Component`** type). You can use the **`Component`** object as a factory to create **`Endpoint`** objects, which in turn act as factories for creating **`Consumer`**, **`Producer`**, and **`Exchange`** objects. These relationships are summarized in Figure 5.1, “Component Factory Patterns”

Figure 5.1. Component Factory Patterns



Component

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

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

Endpoint

Each endpoint instance encapsulates a particular endpoint URI. Every time Apache Camel encounters a new endpoint URI, it creates a new endpoint instance. An endpoint object is also a factory for creating consumer endpoints and producer endpoints.

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

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

Consumer

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

Consumers must implement the **org.apache.camel.Consumer** interface. There are a number of different patterns you can follow when implementing a consumer. These patterns are described in Section 5.1.3, "Consumer Patterns and Threading".

Producer

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

Producers must implement the **org.apache.camel.Producer** interface. You can optionally implement the producer to support an asynchronous style of processing. See Section 5.1.4, "Asynchronous Processing" for details.

Exchange

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

Exchanges must implement the **org.apache.camel.Exchange** interface. The default implementation, **DefaultExchange**, is sufficient for many component implementations. However, if you want to associated extra data with the exchanges or have the exchanges preform additional processing, it can be useful to customize the exchange implementation.

Message

There are two different message slots in an **Exchange** object:

- *In* message—holds the current message.
- *Out* message—temporarily holds a reply message.

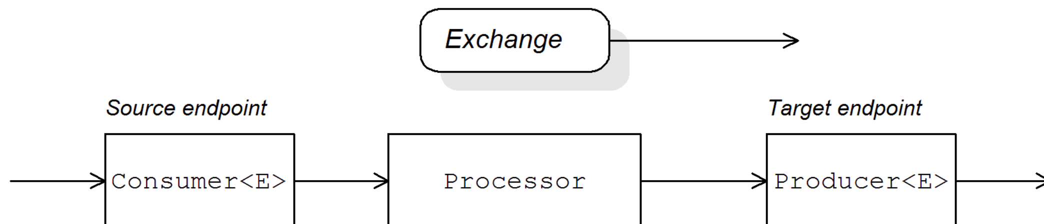
All of the message types are represented by the same Java object, **org.apache.camel.Message**. It is not always necessary to customize the message implementation—the default implementation, **DefaultMessage**, is usually adequate.

5.1.2. Using a Component in a Route

Overview

A Apache Camel route is essentially a pipeline of processors, of **org.apache.camel.Processor** type. Messages are encapsulated in an exchange object, **E**, which gets passed from node to node by invoking the **process()** method. The architecture of the processor pipeline is illustrated in Figure 5.2, "Consumer and Producer Instances in a Route".

Figure 5.2. Consumer and Producer Instances in a Route



Source endpoint

At the start of the route, you have the source endpoint, which is represented by an **org.apache.camel.Consumer** object. The source endpoint is responsible for accepting incoming request messages and dispatching replies. When constructing the route, Apache Camel creates the appropriate **Consumer** type based on the component prefix from the endpoint URI, as described in Section 5.1.1, "Factory Patterns for a Component".

Processors

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

Target endpoint

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

5.1.3. Consumer Patterns and Threading

Overview

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

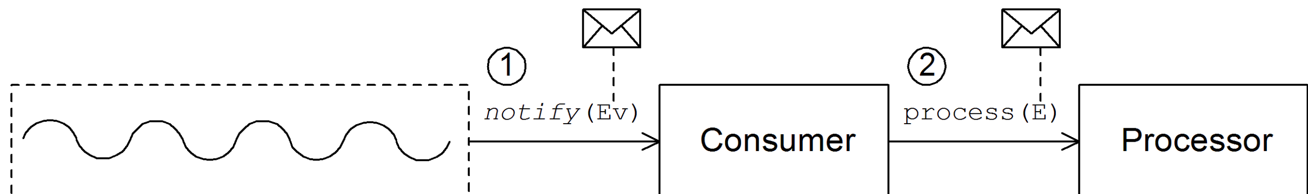
- Event-driven pattern—The consumer is driven by an external thread.
- Scheduled poll pattern—The consumer is driven by a dedicated thread pool.
- Polling pattern—The threading model is left undefined.

Event-driven pattern

In the event-driven pattern, the processing of an incoming request is initiated when another part of the application (typically a third-party library) calls a method implemented by the consumer. A good example of an event-driven consumer is the Apache Camel JMX component, where events are initiated by the JMX library. The JMX library calls the **`handleNotification()`** method to initiate request processing —see Example 8.4, "JMXConsumer Implementation" for details.

Figure 5.3, "Event-Driven Consumer" 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 5.3. Event-Driven Consumer



The event-driven consumer processes incoming requests as follows:

1. The consumer must implement a method to receive the incoming event (in Figure 5.3, "Event-Driven Consumer" this is represented by the ***`notify()`*** method). The thread that calls ***`notify()`*** is normally a separate part of the application, so the consumer's threading policy is externally driven.

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

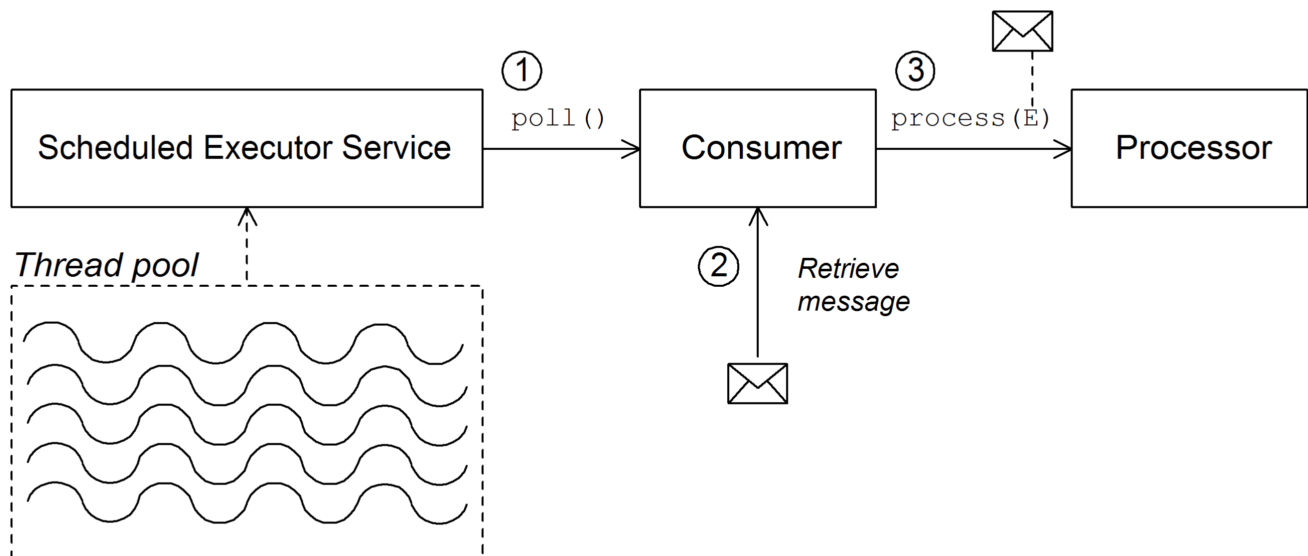
2. In the body of the ***`notify()`*** method, the consumer first converts the incoming event into an exchange object, **`E`**, and then calls **`process()`** on the next processor in the route, passing the exchange object as its argument.

Scheduled poll pattern

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

Figure 5.4, "Scheduled Poll Consumer" shows an outline of the scheduled poll consumer pattern.

Figure 5.4. Scheduled Poll Consumer



The scheduled poll consumer processes incoming requests as follows:

1. The scheduled executor service has a pool of threads at its disposal, that can be used to initiate consumer processing. After each scheduled time interval has elapsed, the scheduled executor service attempts to grab a free thread from its pool (there are five threads in the pool by default). If a free thread is available, it uses that thread to call the **`poll()`** method on the consumer.

2. The consumer's **`poll()`** method is intended to trigger processing of an incoming request. In the body of the **`poll()`** method, the consumer attempts to retrieve an incoming message. If no request is available, the **`poll()`** method returns immediately.

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

Polling pattern

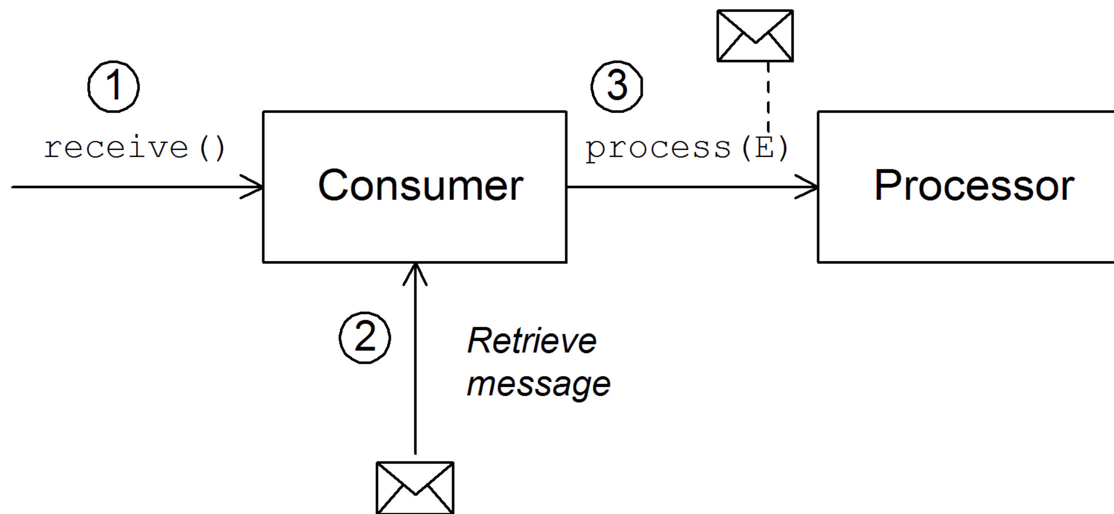
In the polling pattern, processing of an incoming request is initiated when a third-party calls one of the consumer's polling methods:

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

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

Figure 5.5, "Polling Consumer" shows an outline of the polling consumer pattern.

Figure 5.5. Polling Consumer



The polling consumer processes incoming requests as follows:

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

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

   - **`receiveNoWait()`** returns immediately
   - **`receive(long timeout)`** waits for the specified timeout interval[2] before returning
   - **`receive()`** waits until a message is received

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

5.1.4. Asynchronous Processing

Overview

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

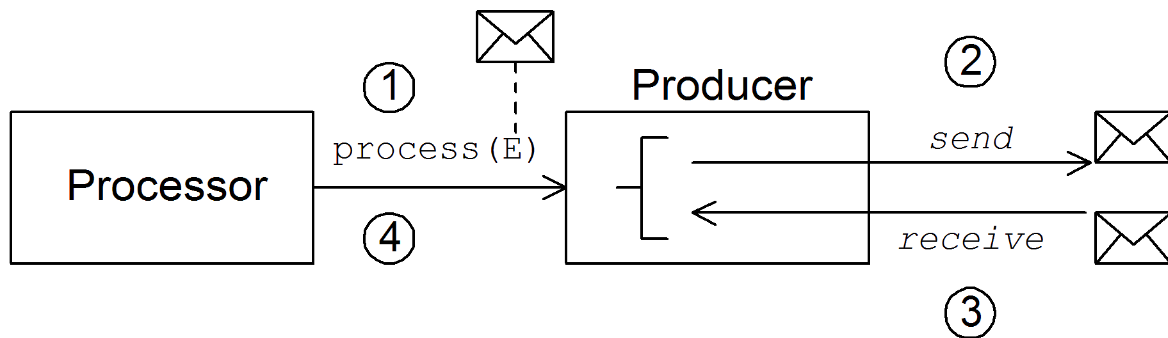
Sometimes, however, you might prefer to decouple the preceding processor from the producer, so that the processor's thread is released immediately and the **`process()`** call does *not* block. In this case, you should implement the producer using an *asynchronous* pattern, which gives the preceding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

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

Synchronous producer

Figure 5.6, “Synchronous Producer” shows an outline of a synchronous producer, where the preceding processor blocks until the producer has finished processing the exchange.

Figure 5.6. Synchronous Producer



The synchronous producer processes an exchange as follows:

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

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

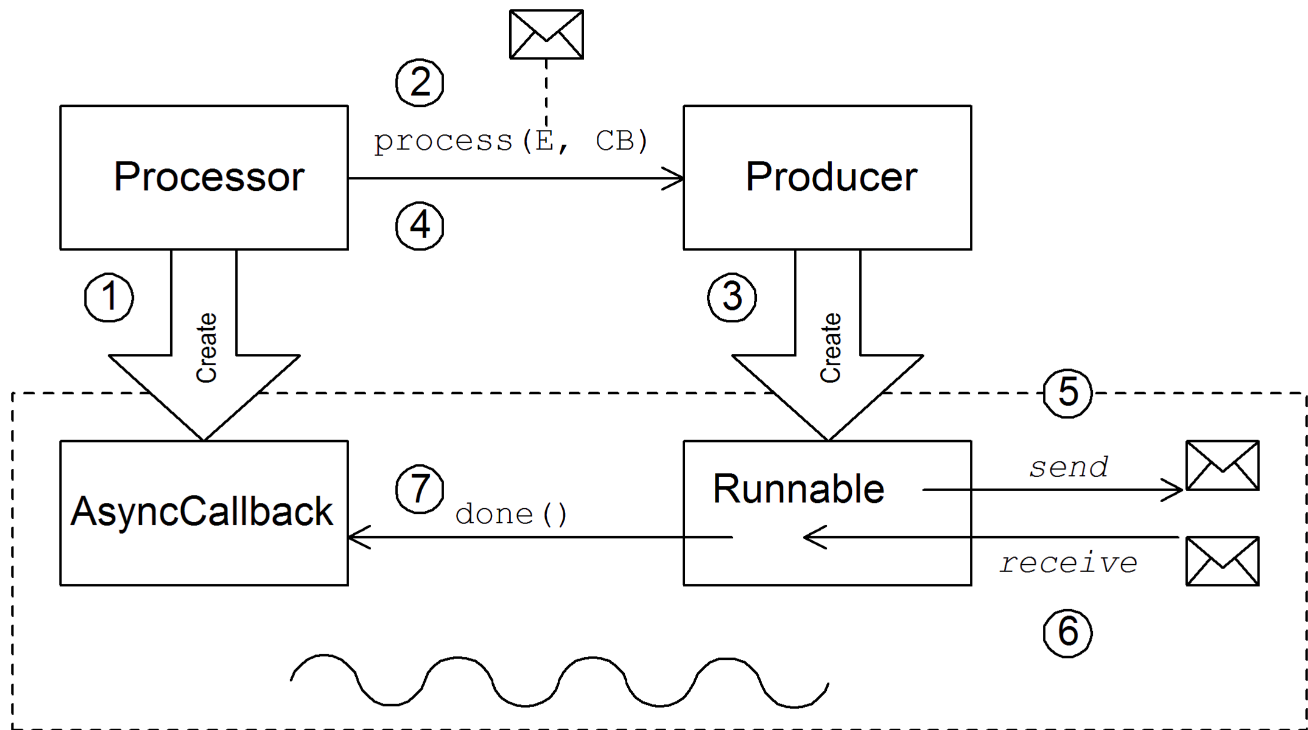
3. If required by the exchange pattern, the producer waits for the reply (*Out* message) to arrive from the endpoint. This step can cause the **`process()`** method to block indefinitely. However, if the exchange pattern does not mandate a reply, the **`process()`** method can return immediately after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

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

Asynchronous producer

Figure 5.7, “Asynchronous Producer” 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 5.7. Asynchronous Producer



The asynchronous producer processes an exchange as follows:

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

2. The processor calls the asynchronous **`process()`** method on the producer to initiate asynchronous processing. The asynchronous **`process()`** method takes two arguments:

   - an exchange object

   - a synchronous callback object

3. In the body of the **`process()`** method, the producer creates a **`Runnable`** object that encapsulates the processing code. The producer then delegates the execution of this **`Runnable`** object to a sub-thread.

4. The asynchronous **`process()`** method returns, thereby freeing up the processor's thread. The exchange processing continues in a separate sub-thread.

5. The **`Runnable`** object sends the *In* message to the endpoint.

6. If required by the exchange pattern, the **`Runnable`** object waits for the reply (*Out* or *Fault* message) to arrive from the endpoint. The **`Runnable`** object remains blocked until the reply is received.

7. After the reply arrives, the **`Runnable`** object inserts the reply (*Out* 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).

5.2. HOW TO IMPLEMENT A COMPONENT

Overview

This section gives a brief overview of the steps required to implement a custom Apache Camel component.

Which interfaces do you need to implement?

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

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

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

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

Implementation steps

You typically implement a custom component as follows:

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

   See Chapter 6, *Component Interface*.

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

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

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

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

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

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

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

See Chapter 7, *Endpoint Interface*.

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

   See Section 8.2, “Implementing the Consumer Interface”.

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

   See Chapter 9, *Producer Interface*.

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

   See Chapter 10, *Exchange Interface* and Chapter 11, *Message Interface*.

Installing and configuring the component

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

- *Add the component directly to the CamelContext*—The **`CamelContext.addComponent()`** method adds a component programatically.
- *Add the component using Spring configuration*—The standard Spring **`bean`** element creates a component instance. The bean's **`id`** attribute implicitly defines the component prefix. For details, see Section 5.3.2, “Configuring a Component”.
- *Configure Apache Camel to auto-discover the component*—Auto-discovery, ensures that Apache Camel automatically loads the component on demand. For details, see Section 5.3.1, “Setting Up Auto-Discovery”.

5.3. AUTO-DISCOVERY AND CONFIGURATION

5.3.1. Setting Up Auto-Discovery

Overview

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

Availability of component classes

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

Configuring auto-discovery

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

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

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

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

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

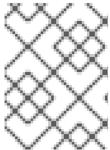
Example

For example, you can enable auto-discovery for the Apache Camel 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
```



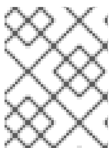
NOTE

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

5.3.2. Configuring a Component

Overview

You can add a component by configuring it in the Apache Camel 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, Apache Camel instantiates the referenced class and injects the properties specified in the Spring configuration.



NOTE

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

Define bean properties on your component class

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

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

```
    void setProperty(PropType v) { ...  }
}
```

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

Configure the component in Spring

To configure a component in Spring, edit the configuration file, **META-INF/spring/camel-context.xml**, as shown in Example 5.1, "Configuring a Component in Spring".

Example 5.1. Configuring a Component in Spring

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

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

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

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

Examples

Example 5.2, "JMS Component Spring Configuration" shows an example of how to configure the Apache Camel's JMS component by defining a bean element with ID equal to **jms**. These settings are added to the Spring configuration file, **camel-context.xml**.

Example 5.2. JMS Component Spring Configuration

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

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

  <bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
2 3     <property name="connectionFactory">
          <bean class="org.apache.activemq.ActiveMQConnectionFactory">
       <property name="brokerURL"
                 value="vm://localhost?
4 broker.persistent=false&amp;broker.useJmx=false"/>
         </bean>
    </property>
  </bean>
</beans>
```

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

2. 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, Apache Camel automatically loads the JMS component using the settings from the **jms** bean element.

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

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

---

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

CHAPTER 6. COMPONENT INTERFACE

Abstract

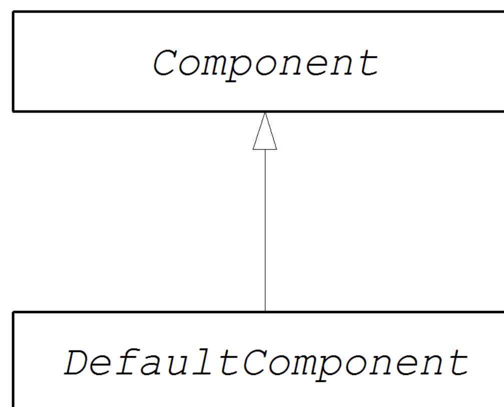
This chapter describes how to implement the **Component** interface.

6.1. THE COMPONENT INTERFACE

Overview

To implement a Apache Camel 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 6.1, "Component Inheritance Hierarchy" shows the relevant Java interfaces and classes that make up the **Component** inheritance hierarchy.

Figure 6.1. Component Inheritance Hierarchy



The Component interface

Example 6.1, "Component Interface" shows the definition of the **org.apache.camel.Component** interface.

Example 6.1. Component Interface

```
package org.apache.camel;

public interface Component {
    CamelContext getCamelContext();
    void setCamelContext(CamelContext context);

    Endpoint createEndpoint(String uri) throws Exception;
}
```

Component methods

The **Component** interface defines the following methods:

- **getCamelContext()** and **setCamelContext()**—References the **CamelContext** to which this **Component** belongs. The **setCamelContext()** method is automatically called when you add the component to a **CamelContext**.

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

6.2. IMPLEMENTING THE COMPONENT INTERFACE

The DefaultComponent class

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

URI parsing

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

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

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

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

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

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

Argument	Sample Value
uri	file:///tmp/messages/foo?delete=true&moveNamePostfix=.old
remaining	**/tmp/messages/foo**

Argument	Sample Value
parameters	Two entries are set in **java.util.Map**: • parameter **delete** is boolean **true** • parameter **moveNamePostfix** has the string value, **.old**.

Parameter injection

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

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

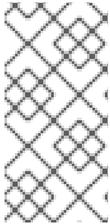
```
public class FileEndpoint extends ScheduledPollEndpoint {
    ...
    public boolean isDelete() {
        return delete;
    }
    public void setDelete(boolean delete) {
        this.delete = delete;
    }
    ...
    public String getMoveNamePostfix() {
        return moveNamePostfix;
    }
  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

It is also possible to inject URI query options into *consumer* parameters. For details, see the section called "Consumer parameter injection".

Disabling endpoint parameter injection

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

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



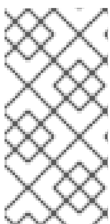
NOTE

The **useIntrospectionOnEndpoint()** method does *not* affect the parameter injection that might be performed on a **Consumer** class. Parameter injection at that level is controlled by the **Endpoint.configureProperties()** method (see Section 7.2, “Implementing the Endpoint Interface”).

Scheduled executor service

The scheduled executor is used in the scheduled poll pattern, where it is responsible for driving the periodic polling of a consumer endpoint (a scheduled executor is effectively a thread pool implementation).

To instantiate a scheduled executor service, use the **ExecutorServiceStrategy** object that is returned by the **CamelContext.getExecutorServiceStrategy()** method. For details of the Apache Camel threading model, see section "Threading Model" in "Implementing Enterprise Integration Patterns".



NOTE

Prior to Apache Camel 2.3, the **DefaultComponent** class provided a **getExecutorService()** method for creating thread pool instances. Since 2.3, however, the creation of thread pools is now managed centrally by the **ExecutorServiceStrategy** object.

Validating the URI

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

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

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

Creating an endpoint

Example 6.2, “Implementation of **createEndpoint()**” outlines how to implement the **DefaultComponent.createEndpoint()** method, which is responsible for creating endpoint instances on demand.

Example 6.2. Implementation of createEndpoint()

```
(1) public class CustomComponent extends DefaultComponent {
        ...
     protected Endpoint createEndpoint(String uri, String remaining, Map
(2) parameters) throws Exception {
(3)         CustomEndpoint result = new CustomEndpoint(uri, this);
            // ...
         return result;
     }
```

```
}
```

1. The *CustomComponent* is the name of your custom component class, which is defined by extending the **`DefaultComponent`** class.

2. When extending **`DefaultComponent`**, you must implement the **`createEndpoint()`** method with three arguments (see the section called “URI parsing”).

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

Example

Example 6.3, “FileComponent Implementation” shows a sample implementation of a **`FileComponent`** class.

Example 6.3. FileComponent Implementation

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

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

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

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

1      public FileComponent() {
       }

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

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

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

2.

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

3 The implementation of the **FileComponent.createEndpoint()** method follows the pattern described in Example 6.2, “Implementation of **createEndpoint()**”. The implementation creates a **FileEndpoint** object.

CHAPTER 7. ENDPOINT INTERFACE

Abstract

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

7.1. THE ENDPOINT INTERFACE

Overview

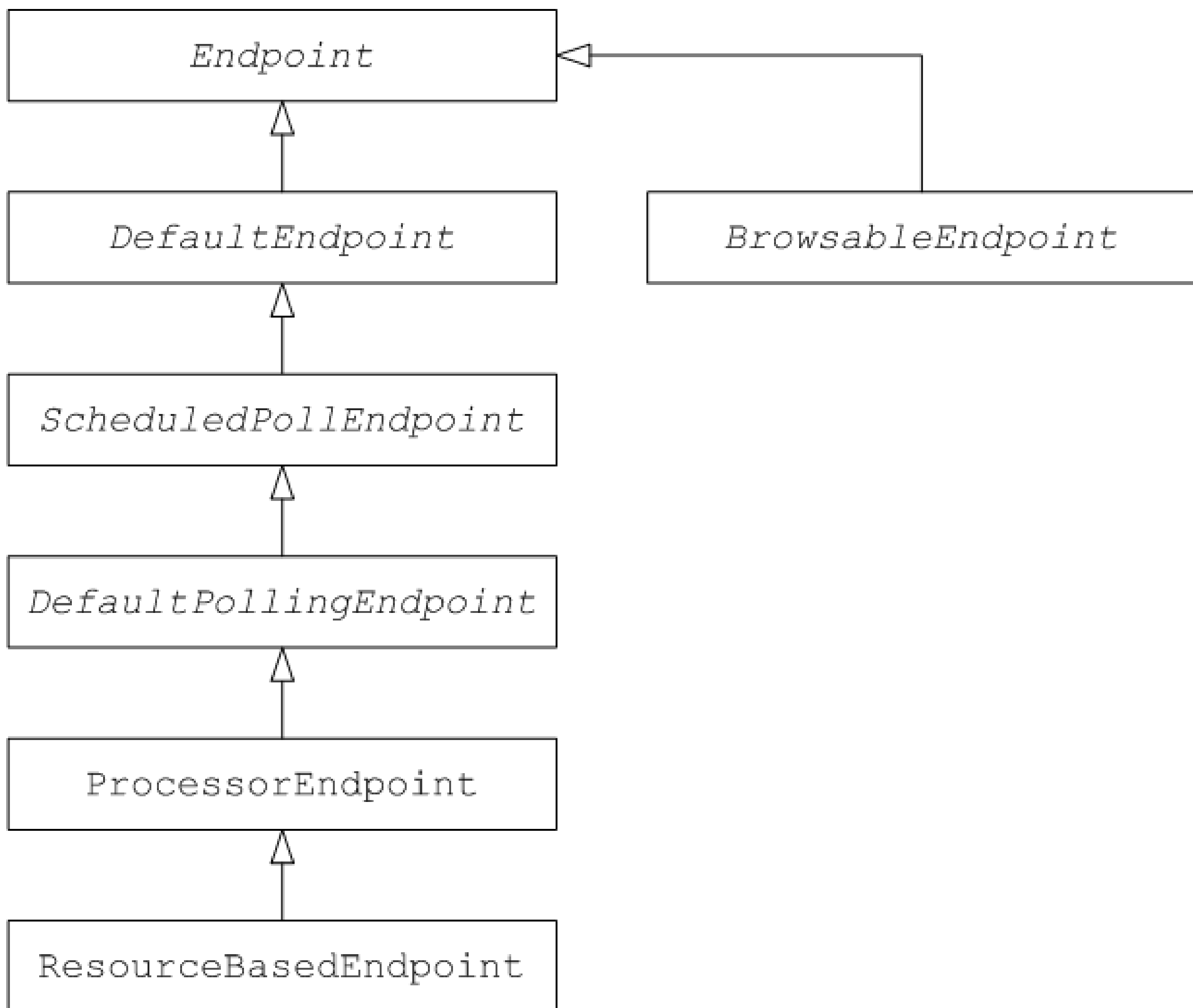
An instance of `org.apache.camel.Endpoint` type encapsulates an endpoint URI, and it also serves as a factory for `Consumer`, `Producer`, and `Exchange` objects. There are three different approaches to implementing an endpoint:

- Event-driven
- scheduled poll
- polling

These endpoint implementation patterns complement the corresponding patterns for implementing a consumer—see Section 8.2, “Implementing the Consumer Interface”.

Figure 7.1, “Endpoint Inheritance Hierarchy” shows the relevant Java interfaces and classes that make up the `Endpoint` inheritance hierarchy.

Figure 7.1. Endpoint Inheritance Hierarchy



The Endpoint interface

Example 7.1, “Endpoint Interface” shows the definition of the **`org.apache.camel.Endpoint`** interface.

Example 7.1. Endpoint Interface

```
package org.apache.camel;

public interface Endpoint {
    boolean isSingleton();

    String getEndpointUri();

    String getEndpointKey();

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

    void configureProperties(Map options);

    boolean isLenientProperties();

    Exchange createExchange();
```

```
    Exchange createExchange(ExchangePattern pattern);
    Exchange createExchange(Exchange exchange);

    Producer createProducer() throws Exception;

    Consumer createConsumer(Processor processor) throws Exception;
    PollingConsumer createPollingConsumer() throws Exception;
}
```

Endpoint methods

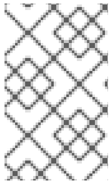
The **`Endpoint`** interface defines the following methods:

- **`isSingleton()`**—Returns **`true`**, if you want to ensure that each URI maps to a single endpoint within a CamelContext. When this property is **`true`**, multiple references to the identical URI within your routes always refer to a *single* endpoint instance. When this property is **`false`**, on the other hand, multiple references to the same URI within your routes refer to *distinct* endpoint instances. Each time you refer to the URI in a route, a new endpoint instance is created.
- **`getEndpointUri()`**—Returns the endpoint URI of this endpoint.
- **`getEndpointKey()`**—Used by **`org.apache.camel.spi.LifecycleStrategy`** when registering the endpoint.
- **`getCamelContext()`**—return a reference to the **`CamelContext`** instance to which this endpoint belongs.
- **`setCamelContext()`**—Sets the **`CamelContext`** instance to which this endpoint belongs.
- **`configureProperties()`**—Stores a copy of the parameter map that is used to inject parameters when creating a new **`Consumer`** instance.
- **`isLenientProperties()`**—Returns **`true`** to indicate that the URI is allowed to contain unknown parameters (that is, parameters that cannot be injected on the **`Endpoint`** or the **`Consumer`** class). Normally, this method should be implemented to return **`false`**.
- **`createExchange()`**—An overloaded method with the following variants:
  - **`Exchange createExchange()`**—Creates a new exchange instance with a default exchange pattern setting.
  - **`Exchange createExchange(ExchangePattern pattern)`**—Creates a new exchange instance with the specified exchange pattern.
  - **`Exchange createExchange(Exchange exchange)`**—Converts the given **`exchange`** argument to the type of exchange needed for this endpoint. If the given exchange is not already of the correct type, this method copies it into a new instance of the correct type. A default implementation of this method is provided in the **`DefaultEndpoint`** class.
- **`createProducer()`**—Factory method used to create new **`Producer`** instances.
- **`createConsumer()`**—Factory method to create new event-driven consumer instances. The **`processor`** argument is a reference to the first processor in the route.

- **createPollingConsumer()**—Factory method to create new polling consumer instances.

Endpoint singletons

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



NOTE

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

7.2. IMPLEMENTING THE ENDPOINT INTERFACE

Alternative ways of implementing an endpoint

The following alternative endpoint implementation patterns are supported:

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

Event-driven endpoint implementation

If your custom endpoint conforms to the event-driven pattern (see Section 5.1.3, “Consumer Patterns and Threading”), it is implemented by extending the abstract class, **org.apache.camel.impl.DefaultEndpoint**, as shown in Example 7.2, “Implementing DefaultEndpoint”.

Example 7.2. Implementing DefaultEndpoint

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

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

(1) public class CustomEndpoint extends DefaultEndpoint {

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

        public Producer createProducer() throws Exception {
```

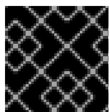
```
3       return new CustomProducer(this);
    }

    public Consumer createConsumer(Processor processor) throws Exception
4{
            return new CustomConsumer(this, processor);
    }

    public boolean isSingleton() {
        return true;
    }

    // Implement the following methods, only if you need to set exchange
properties.
    //
5     public Exchange createExchange() {
            return this.createExchange(getExchangePattern());
    }

    public Exchange createExchange(ExchangePattern pattern) {
        Exchange result = new DefaultExchange(getCamelContext(),
pattern);
        // Set exchange properties
        ...
        return result;
    }
}
```

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

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

3. Implement the **createProducer()** factory method to create producer endpoints.

4. Implement the **createConsumer()** factory method to create event-driven consumer instances.



IMPORTANT

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

5. In general, it is *not* necessary to override the **createExchange()** methods. The implementations inherited from **DefaultEndpoint** create a **DefaultExchange** object by default, which can be used in any Apache Camel component. If you need to initialize some exchange properties in the **DefaultExchange** object, however, it is appropriate to override the **createExchange()** methods here in order to add the exchange property settings.

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

- **getEndpointUri()**—Returns the endpoint URI.

- **getCamelContext()**—Returns a reference to the **CamelContext**.
- **getComponent()**—Returns a reference to the parent component.
- **createPollingConsumer()**—Creates a polling consumer. The created polling consumer's functionality is based on the event-driven consumer. If you override the event-driven consumer method, **createConsumer()**, you get a polling consumer implementation for free.
- **createExchange(Exchange e)**—Converts the given exchange object, **e**, to the type required for this endpoint. This method creates a new endpoint using the overridden **createExchange()** endpoints. This ensures that the method also works for custom exchange types.

Scheduled poll endpoint implementation

If your custom endpoint conforms to the scheduled poll pattern (see Section 5.1.3, “Consumer Patterns and Threading”) it is implemented by inheriting from the abstract class, **org.apache.camel.impl.ScheduledPollEndpoint**, as shown in Example 7.3, “ScheduledPollEndpoint Implementation”.

Example 7.3. ScheduledPollEndpoint Implementation

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

1 public class CustomEndpoint extends ScheduledPollEndpoint {

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

3     public Producer createProducer() throws Exception {
        Producer result = new CustomProducer(this);
      return result;
    }

    public Consumer createConsumer(Processor processor) throws Exception
4 {
        Consumer result = new CustomConsumer(this, processor);
5       configureConsumer(result);
        return result;
    }

    public boolean isSingleton() {
        return true;
    }

    // Implement the following methods, only if you need to set exchange
properties.
```

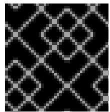
```
    //
6     public Exchange createExchange() {
          return this.createExchange(getExchangePattern());
    }

    public Exchange createExchange(ExchangePattern pattern) {
        Exchange result = new DefaultExchange(getCamelContext(),
pattern);
        // Set exchange properties
        ...
        return result;
    }
}
```

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

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

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

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



IMPORTANT

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

5. The **`configureConsumer()`** method, defined in the **`ScheduledPollEndpoint`** base class, is responsible for injecting consumer query options into the consumer. See the section called "Consumer parameter injection".

6. In general, it is *not* necessary to override the **`createExchange()`** methods. The implementations inherited from **`DefaultEndpoint`** create a **`DefaultExchange`** object by default, which can be used in any Apache Camel component. If you need to initialize some exchange properties in the **`DefaultExchange`** object, however, it is appropriate to override the **`createExchange()`** methods here in order to add the exchange property settings.

Polling endpoint implementation

If your custom endpoint conforms to the polling consumer pattern (see Section 5.1.3, "Consumer Patterns and Threading"), it is implemented by inheriting from the abstract class, **`org.apache.camel.impl.DefaultPollingEndpoint`**, as shown in Example 7.4, "DefaultPollingEndpoint Implementation".

Example 7.4. DefaultPollingEndpoint Implementation

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

```
import org.apache.camel.impl.DefaultPollingEndpoint;

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

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

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

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 7.3, "ScheduledPollEndpoint Implementation" for details.

Implementing the BrowsableEndpoint interface

If you want to expose the list of exchange instances that are pending in the current endpoint, you can implement the **`org.apache.camel.spi.BrowsableEndpoint`** interface, as shown in Example 7.5, "BrowsableEndpoint Interface". It makes sense to implement this interface if the endpoint performs some sort of buffering of incoming events. For example, the Apache Camel SEDA endpoint implements the **`BrowsableEndpoint`** interface—see Example 7.6, "SedaEndpoint Implementation".

Example 7.5. BrowsableEndpoint Interface

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

import java.util.List;

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

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

Example

Example 7.6, "SedaEndpoint Implementation" shows a sample implementation of **`SedaEndpoint`**. 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 7.6. SedaEndpoint Implementation

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

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

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

public class SedaEndpoint extends DefaultEndpoint implements
1 BrowsableEndpoint {
      private BlockingQueue<Exchange> queue;

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

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

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

    public Consumer createConsumer(Processor processor) throws Exception
5 {
          return new SedaConsumer(this, processor);
    }

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

7     public boolean isSingleton() {
          return true;
    }

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

1. 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.
2. 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**.
3. Another constructor is provided, which delegates queue creation to the parent component instance.
4. The **createProducer()** factory method creates an instance of **CollectionProducer**, which is a producer implementation that adds events to the queue.
5. The **createConsumer()** factory method creates an instance of **SedaConsumer**, which is responsible for pulling events off the queue and processing them.
6. The **getQueue()** method returns a reference to the queue.
7. The **isSingleton()** method returns **true**, indicating that a single endpoint instance should be created for each unique URI string.
8. The **getExchanges()** method implements the corresponding abstract method from **BrowsableEndpoint**.

CHAPTER 8. CONSUMER INTERFACE

Abstract

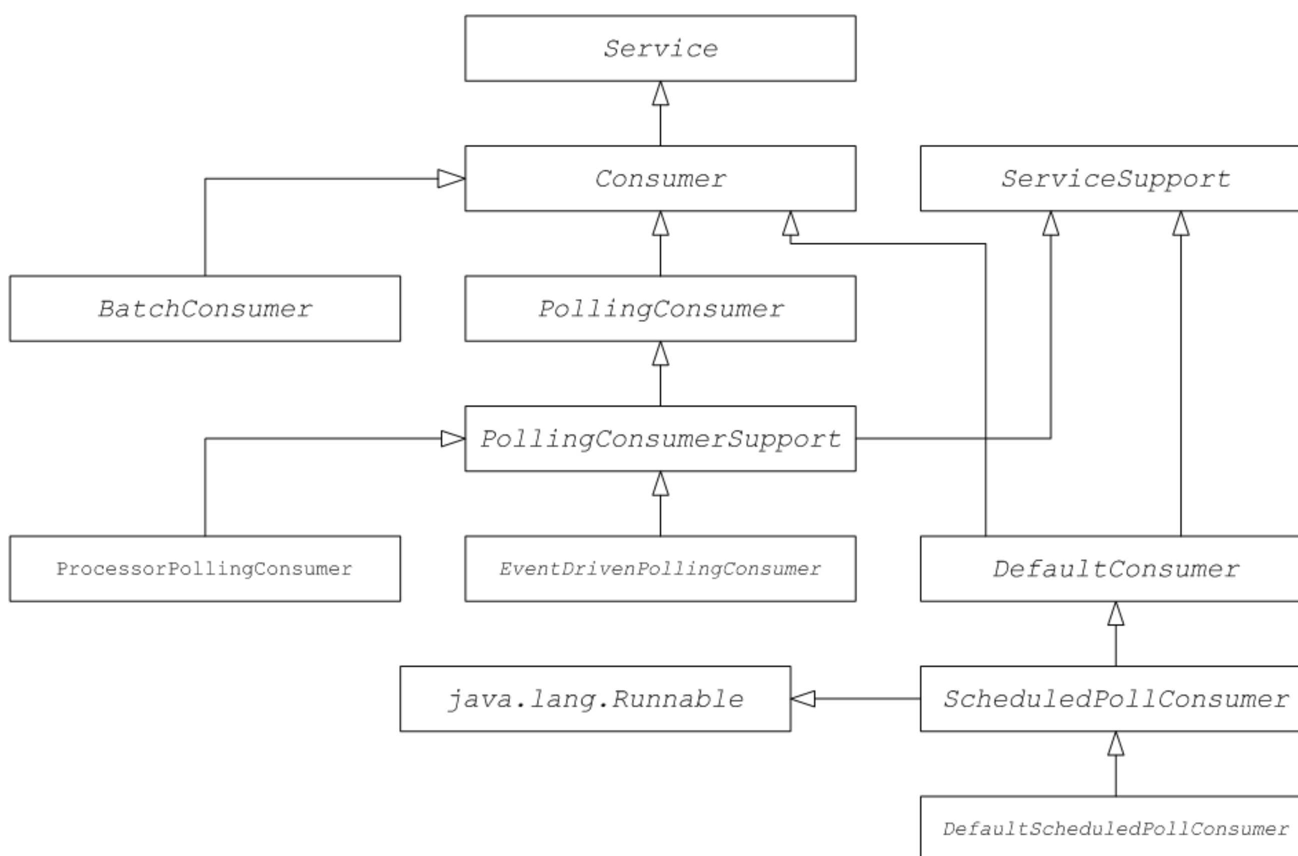
This chapter describes how to implement the **Consumer** interface, which is an essential step in the implementation of a Apache Camel component.

8.1. 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 Section 5.1.3, “Consumer Patterns and Threading”), and this degree of flexibility is reflected in the inheritance hierarchy (see Figure 8.1, “Consumer Inheritance Hierarchy”), which includes several different base classes for implementing a consumer.

Figure 8.1. Consumer Inheritance Hierarchy



Consumer parameter injection

For consumers that follow the scheduled poll pattern (see the section called “Scheduled poll pattern”), Apache Camel 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
```

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

```
public class CustomConsumer extends ScheduledPollConsumer {
    ...
    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 the section called "Scheduled poll endpoint implementation"). Example 8.1, "FileEndpoint createConsumer() Implementation" shows an example of a **createConsumer()** method implementation, taken from the **FileEndpoint** class in the file component:

Example 8.1. FileEndpoint createConsumer() Implementation

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

At run time, consumer parameter injection works as follows:

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

2. When **createConsumer()** is called, the method implementation calls **configureConsumer()** to inject the consumer parameters (see Example 8.1, "FileEndpoint createConsumer() Implementation").

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 8.1, "Scheduled Poll Parameters" (which can appear as query options in the endpoint URI).

Table 8.1. Scheduled Poll Parameters

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

Converting between event-driven and polling consumers

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

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

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

```
package org.apache.camel;

public interface Endpoint {
    ...
    Consumer createConsumer(Processor processor) throws Exception;
    PollingConsumer createPollingConsumer() throws Exception;
}
```

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

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

where it is defined as follows:

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

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

Analogously, if you are implementing your consumer according to the polling pattern, you implement the endpoint by extending **DefaultPollingEndpoint** and implementing the **createPollingConsumer()** method. In this case, the implementation of the **createConsumer()** method is inherited from **DefaultPollingEndpoint**, and the default implementation returns a **DefaultScheduledPollConsumer** instance (which converts the polling consumer into an event-driven consumer).

ShutdownPrepared interface

Consumer classes can optionally implement the **org.apache.camel.spi.ShutdownPrepared** interface, which enables your custom consumer endpoint to receive shutdown notifications.

Example 8.2, "ShutdownPrepared Interface" shows the definition of the **ShutdownPrepared** interface.

Example 8.2. ShutdownPrepared Interface

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


public interface ShutdownPrepared {

    void prepareShutdown(boolean forced);

}
```

The **ShutdownPrepared** interface defines the following methods:

prepareShutdown

Receives notifications to shut down the consumer endpoint in one or two phases, as follows:

1. *Graceful shutdown*—where the **forced** argument has the value **false**. Attempt to clean up resources gracefully. For example, by stopping threads gracefully.
2. *Forced shutdown*—where the **forced** argument has the value **true**. This means that the shutdown has timed out, so you must clean up resources more aggressively. This is the last chance to clean up resources before the process exits.

ShutdownAware interface

Consumer classes can optionally implement the **org.apache.camel.spi.ShutdownAware** interface, which interacts with the graceful shutdown mechanism, enabling a consumer to ask for extra time to shut down. This is typically needed for components such as SEDA, which can have pending exchanges stored in an internal queue. Normally, you would want to process all of the exchanges in the queue before shutting down the SEDA consumer.

Example 8.3, "ShutdownAware Interface" shows the definition of the **ShutdownAware** interface.

Example 8.3. ShutdownAware Interface

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

import org.apache.camel.ShutdownRunningTask;

public interface ShutdownAware extends ShutdownPrepared {

    boolean deferShutdown(ShutdownRunningTask shutdownRunningTask);

    int getPendingExchangesSize();
}
```

The **ShutdownAware** interface defines the following methods:

deferShutdown

Return **true** from this method, if you want to delay shutdown of the consumer. The **shutdownRunningTask** argument is an **enum** which can take either of the following values:

- **ShutdownRunningTask.CompleteCurrentTaskOnly**—finish processing the exchanges that are currently being processed by the consumer's thread pool, but do not attempt to process any more exchanges than that.
- **ShutdownRunningTask.CompleteAllTasks**—process *all* of the pending exchanges. For example, in the case of the SEDA component, the consumer would process all of the exchanges from its incoming queue.

getPendingExchangesSize

Indicates how many exchanges remain to be processed by the consumer. A zero value indicates that processing is finished and the consumer can be shut down.

For an example of how to define the **ShutdownAware** methods, see Example 8.7, "Custom Threading Implementation".

8.2. IMPLEMENTING THE CONSUMER INTERFACE

Alternative ways of implementing a consumer

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

- Event-driven consumer implementation

- Scheduled poll consumer implementation
- Polling consumer implementation
- Custom threading implementation

Event-driven consumer implementation

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

Example 8.4, "JMXConsumer Implementation" shows the implementation of the **JMXConsumer** class, which is taken from the Apache Camel JMX component implementation. The **JMXConsumer** class is an example of an event-driven consumer, which is implemented by inheriting from the **org.apache.camel.impl.DefaultConsumer** class. In the case of the **JMXConsumer** example, events are represented by calls on the **NotificationListener.handleNotification()** method, which is a standard way of receiving JMX events. In order to receive these JMX events, it is necessary to implement the **NotificationListener** interface and override the **handleNotification()** method, as shown in Example 8.4, "JMXConsumer Implementation".

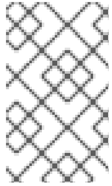
Example 8.4. JMXConsumer Implementation

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

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

public class JMXConsumer extends DefaultConsumer implements
NotificationListener { 1

    JMXEndpoint jmxEndpoint;

    public JMXConsumer(JMXEndpoint endpoint, Processor processor) { 2
        super(endpoint, processor);
        this.jmxEndpoint = endpoint;
    }

    public void handleNotification(Notification notification, Object
handback) { 3
        try {
getProcessor().process(jmxEndpoint.createExchange(notification)); 4
        } catch (Throwable e) {
            handleException(e); 5
        }
    }
}
```

1. 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.

2. 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.

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



NOTE

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

4. This line of code combines two steps. First, the JMX notification object is converted into an exchange object, which is the generic representation of an event in Apache Camel. Then the newly created exchange object is passed to the next processor in the route (invoked synchronously).

5. 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 Section 5.1.3, “Consumer Patterns and Threading”).

Example 8.5, “ScheduledPollConsumer Implementation” shows how to implement a consumer that follows the scheduled poll pattern, which is implemented by extending the **ScheduledPollConsumer** class.

Example 8.5. ScheduledPollConsumer Implementation

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

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

import org.apache.camel.impl.ScheduledPollConsumer;

(1) public class CustomConsumer extends ScheduledPollConsumer {
        private final CustomEndpoint endpoint;

      public CustomConsumer(CustomEndpoint endpoint, Processor processor)
(2) {
            super(endpoint, processor);
```

```
        this.endpoint = endpoint;
    }

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

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

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

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

2. 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.

3. Override the **poll()** method to receive the scheduled polling events. This is where you should put the code that retrieves and processes incoming events (represented by exchange objects).

4. 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 Section 5.1.4, “Asynchronous Processing”.

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

6. *(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 8.6, “PollingConsumerSupport Implementation” outlines how to implement a consumer that follows the polling pattern, which is implemented by extending the **PollingConsumerSupport** class.

Example 8.6. PollingConsumerSupport Implementation

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

public class CustomConsumer extends PollingConsumerSupport {
    private final CustomEndpoint endpoint;

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

    public Exchange receiveNoWait() {
        Exchange exchange = /* Obtain an exchange object. */;
      // Further processing ...
      return exchange;
    }

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

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

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

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

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

2. 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.

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

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

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

6. 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.

Custom threading implementation

If the standard consumer patterns are not suitable for your consumer implementation, you can implement the **Consumer** interface directly and write the threading code yourself. When writing the threading code, however, it is important that you comply with the standard Apache Camel threading model, as described in section "Threading Model" in "Implementing Enterprise Integration Patterns".

For example, the SEDA component from **camel-core** implements its own consumer threading, which is consistent with the Apache Camel threading model. Example 8.7, "Custom Threading Implementation" shows an outline of how the **SedaConsumer** class implements its threading.

Example 8.7. Custom Threading Implementation

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

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

import org.apache.camel.Consumer;
import org.apache.camel.Endpoint;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.ShutdownRunningTask;
import org.apache.camel.impl.LoggingExceptionHandler;
import org.apache.camel.impl.ServiceSupport;
import org.apache.camel.util.ServiceHelper;
...
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * A Consumer for the SEDA component.
 *
 * @version $Revision: 922485 $
 */
public class SedaConsumer extends ServiceSupport implements Consumer,
1 Runnable, ShutdownAware {
      private static final transient Log LOG =
LogFactory.getLog(SedaConsumer.class);

    private SedaEndpoint endpoint;
    private Processor processor;
    private ExecutorService executor;
    ...
    public SedaConsumer(SedaEndpoint endpoint, Processor processor) {
        this.endpoint = endpoint;
        this.processor = processor;
    }
    ...

2     public void run() {
          BlockingQueue<Exchange> queue = endpoint.getQueue();
        // Poll the queue and process exchanges
```

```
            ...
        }

        ...
3       protected void doStart() throws Exception {
            int poolSize = endpoint.getConcurrentConsumers();
            executor =
endpoint.getCamelContext().getExecutorServiceStrategy()
                .newFixedThreadPool(this, endpoint.getEndpointUri(),
4 poolSize);
5           for (int i = 0; i < poolSize; i++) {
                executor.execute(this);
            }
            endpoint.onStarted(this);
        }

6       protected void doStop() throws Exception {
            endpoint.onStopped(this);
            // must shutdown executor on stop to avoid overhead of having
them running

endpoint.getCamelContext().getExecutorServiceStrategy().shutdownNow(exec
7 utor);
            executor = null;

            if (multicast != null) {
                ServiceHelper.stopServices(multicast);
            }
        }
        ...
        //----------
        // Implementation of ShutdownAware interface

        public boolean deferShutdown(ShutdownRunningTask
shutdownRunningTask) {
            // deny stopping on shutdown as we want seda consumers to run
in case some other queues
            // depend on this consumer to run, so it can complete its
exchanges
            return true;
        }

        public int getPendingExchangesSize() {
            // number of pending messages on the queue
            return endpoint.getQueue().size();
        }

}
```

1 The **`SedaConsumer`** class is implemented by extending the **`org.apache.camel.impl.ServiceSupport`** class and implementing the **`Consumer`**, **`Runnable`**, and **`ShutdownAware`** interfaces.

2

Implement the **Runnable.run()** method to define what the consumer does while it is running in a thread. In this case, the consumer runs in a loop, polling the queue for new exchanges and then

3. The **doStart()** method is inherited from **ServiceSupport**. You override this method in order to define what the consumer does when it starts up.

4. Instead of creating threads directly, you should create a thread pool using the **ExecutorServiceStrategy** object that is registered with the **CamelContext**. This is important, because it enables Apache Camel to implement centralized management of threads and support such features as graceful shutdown.

   For details, see section "Threading Model" in "Implementing Enterprise Integration Patterns".

5. Kick off the threads by calling the **ExecutorService.execute()** method **poolSize** times.

6. The **doStop()** method is inherited from **ServiceSupport**. You override this method in order to define what the consumer does when it shuts down.

7. Shut down the thread pool, which is represented by the **executor** instance.

CHAPTER 9. PRODUCER INTERFACE

Abstract

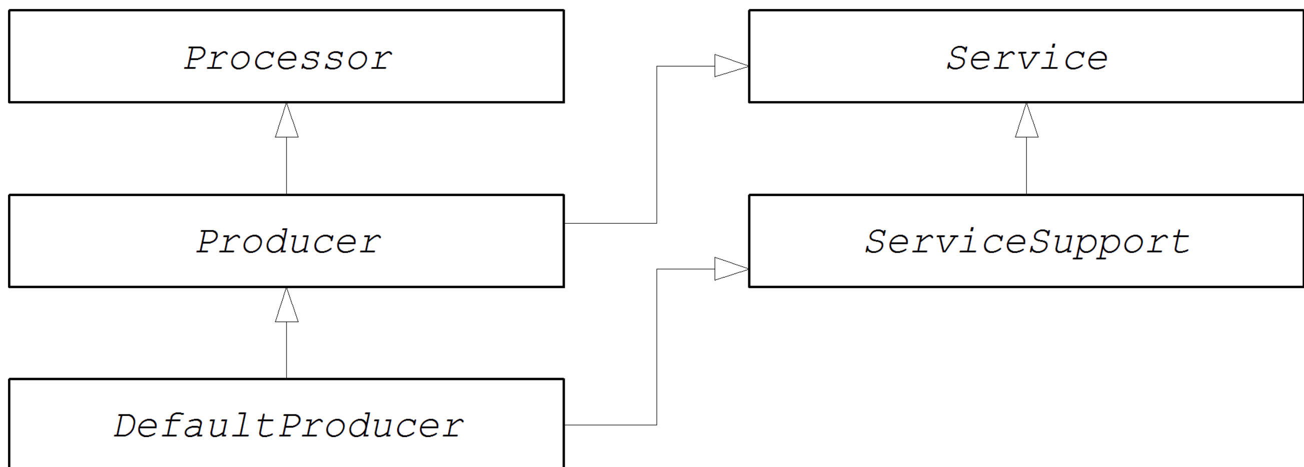
This chapter describes how to implement the **Producer** interface, which is an essential step in the implementation of a Apache Camel component.

9.1. 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 9.1, “Producer Inheritance Hierarchy” shows the inheritance hierarchy for producers.

Figure 9.1. Producer Inheritance Hierarchy



The Producer interface

Example 9.1, “Producer Interface” shows the definition of the **org.apache.camel.Producer** interface.

Example 9.1. Producer Interface

```
package org.apache.camel;

public interface Producer extends Processor, Service, IsSingleton {

    Endpoint<E> getEndpoint();

    Exchange createExchange();

    Exchange createExchange(ExchangePattern pattern);

    Exchange createExchange(E exchange);
}
```

Producer methods

The **Producer** interface defines the following methods:

- **process()** *(inherited from Processor)*—The most important method. A producer is essentially a special type of processor that sends a request to an endpoint, instead of forwarding the exchange object to another processor. By overriding the **process()** method, you define how the producer sends and receives messages to and from the relevant endpoint.

- **getEndpoint()**—Returns a reference to the parent endpoint instance.

- **createExchange()**—These overloaded methods are analogous to the corresponding methods defined in the **Endpoint** interface. Normally, these methods delegate to the corresponding methods defined on the parent **Endpoint** instance (this is what the **DefaultEndpoint** class does by default). Occasionally, you might need to override these methods.

Asynchronous processing

Processing an exchange object in a producer—which usually involves sending a message to a remote destination and waiting for a reply—can potentially block for a significant length of time. If you want to avoid blocking the current thread, you can opt to implement the producer as an *asynchronous processor*. The asynchronous processing pattern decouples the preceding processor from the producer, so that the **process()** method returns without delay. See Section 5.1.4, “Asynchronous Processing”.

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 9.2, “AsyncProcessor Interface”.

Example 9.2. AsyncProcessor Interface

```
package org.apache.camel;

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

The asynchronous version of the **process()** method takes an extra argument, **callback**, of **org.apache.camel.AsyncCallback** type. The corresponding **AsyncCallback** interface is defined as shown in Example 9.3, “AsyncCallback Interface”.

Example 9.3. AsyncCallback Interface

```
package org.apache.camel;

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

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

ExchangeHelper class

When implementing a producer, you might find it helpful to call some of the methods in the **org.apache.camel.util.ExchangeHelper** utility class. For full details of the **ExchangeHelper** class, see Section 2.4, “The ExchangeHelper Class”.

9.2. IMPLEMENTING THE PRODUCER INTERFACE

Alternative ways of implementing a producer

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

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

How to implement a synchronous producer

Example 9.4, “DefaultProducer Implementation” outlines how to implement a synchronous producer. In this case, call to **Producer.process()** blocks until a reply is received.

Example 9.4. DefaultProducer Implementation

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

1 public class CustomProducer extends DefaultProducer {

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

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

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

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

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

   - If the exchange contains an *In* message, and if this is consistent with the specified exchange pattern, then send the *In* message to the designated endpoint.
   - If the exchange pattern anticipates the receipt of an *Out* message, then wait until the *Out* 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()** to attach the reply to the exchange object. If the reply contains a fault message, set the fault flag on the *Out* message using **Message.setFault(true)**.

How to implement an asynchronous producer

Example 9.5, "CollectionProducer Implementation" 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 9.5. CollectionProducer Implementation

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

public class CustomProducer extends DefaultProducer implements
AsyncProcessor { 1

    public CustomProducer(Endpoint endpoint) { 2
        super(endpoint);
      // ...
    }

    public void process(Exchange exchange) throws Exception { 3
        // Process exchange synchronously.
      // ...
    }

    public boolean process(Exchange exchange, AsyncCallback callback) {
        // Process exchange asynchronously. 4
        CustomProducerTask task = new CustomProducerTask(exchange,
callback);
      // Process 'task' in a separate thread...
      // ...
        return false; 5
    }
}
```

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

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

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

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

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

3. Implement the synchronous **`process()`** method.

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

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

6. 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.

7. The **`run()`** method contains the code that sends the *In* message to the producer endpoint and waits to receive the reply, if any. After receiving the reply (*Out* message or *Fault* message) and inserting it into the exchange object, you must call **`callback.done()`** to notify the caller that processing is complete.

CHAPTER 10. EXCHANGE INTERFACE

Abstract

This chapter describes the **Exchange** interface. Since the refactoring of the camel-core module performed in Apache Camel 2.0, there is no longer any necessity to define custom exchange types. The **DefaultExchange** implementation can now be used in all cases.

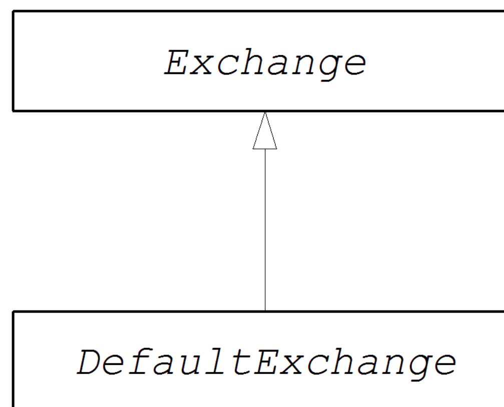
10.1. THE EXCHANGE INTERFACE

Overview

An instance of **org.apache.camel.Exchange** type encapsulates the current message passing through a route, with additional metadata encoded as exchange properties.

Figure 10.1, "Exchange Inheritance Hierarchy" shows the inheritance hierarchy for the exchange type. The default implementation, **DefaultExchange**, is always used.

Figure 10.1. Exchange Inheritance Hierarchy



The Exchange interface

Example 10.1, "Exchange Interface" shows the definition of the **org.apache.camel.Exchange** interface.

Example 10.1. Exchange Interface

```
package org.apache.camel;

import java.util.Map;

import org.apache.camel.spi.Synchronization;
import org.apache.camel.spi.UnitOfWork;

public interface Exchange {
    // Exchange property names (string constants)
    // (Not shown here)
    ...

    ExchangePattern getPattern();
```

```
    void setPattern(ExchangePattern pattern);

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

    Message getIn();
    <T> T   getIn(Class<T> type);
    void    setIn(Message in);

    Message getOut();
    <T> T   getOut(Class<T> type);
    void    setOut(Message out);
    boolean hasOut();

    Throwable getException();
    <T> T     getException(Class<T> type);
    void      setException(Throwable e);

    boolean isFailed();

    boolean isTransacted();

    boolean isRollbackOnly();

    CamelContext getContext();

    Exchange copy();

    Endpoint getFromEndpoint();
    void     setFromEndpoint(Endpoint fromEndpoint);

    String getFromRouteId();
    void   setFromRouteId(String fromRouteId);

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

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

    void addOnCompletion(Synchronization onCompletion);
    void handoverCompletions(Exchange target);
}
```

Exchange methods

The **Exchange** interface defines the following methods:

- **`getPattern()`**, **`setPattern()`**—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`**
- **`setProperty()`**, **`getProperty()`**, **`getProperties()`**, **`removeProperty()`**, **`hasProperties()`**—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 component implementation.
- **`setIn()`**, **`getIn()`**—Setter and getter methods for the *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()`**, **`hasOut()`**—Setter and getter methods for the *Out* message.

  The **`getOut()`** method implicitly supports lazy creation of an *Out* message. That is, if the current *Out* message is **`null`**, a new message instance is automatically created.
- **`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.
- **`isTransacted()`**—Returns **`true`**, if the exchange is transacted.
- **`isRollback()`**—Returns **`true`**, if the exchange is marked for rollback.
- **`getContext()`**—Returns a reference to the associated **`CamelContext`** instance.
- **`copy()`**—Creates a new, identical (apart from the exchange ID) copy of the current custom exchange object. The body and headers of the *In* message, the *Out* message (if any), and the *Fault* message (if any) are also copied by this operation.
- **`setFromEndpoint()`**, **`getFromEndpoint()`**—Getter and setter methods for the consumer endpoint that orginated this message (which is typically the endpoint appearing in the **`from()`** DSL command at the start of a route).

- **`setFromRouteId()`**, **`getFromRouteId()`**—Getters and setters for the route ID that originated this exchange. The **`getFromRouteId()`** method should only be called internally.

- **`setUnitOfWork()`**, **`getUnitOfWork()`**—Getter and setter methods for the **`org.apache.camel.spi.UnitOfWork`** bean property. This property is only required for exchanges that can participate in a transaction.

- **`setExchangeId()`**, **`getExchangeId()`**—Getter and setter methods for the exchange ID. Whether or not a custom component uses and exchange ID is an implementation detail.

- **`addOnCompletion()`**—Adds an **`org.apache.camel.spi.Synchronization`** callback object, which gets called when processing of the exchange has completed.

- **`handoverCompletions()`**—Hands over all of the *OnCompletion* callback objects to the specified exchange object.

CHAPTER 11. MESSAGE INTERFACE

Abstract

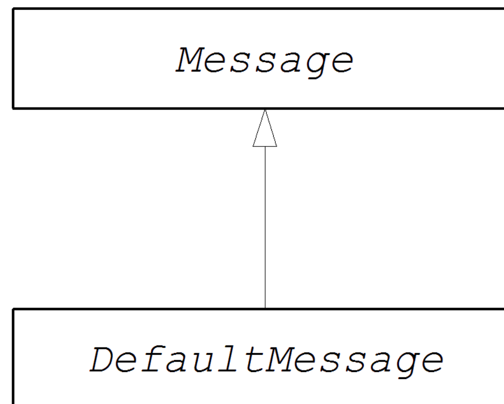
This chapter describes how to implement the **Message** interface, which is an optional step in the implementation of a Apache Camel component.

11.1. THE MESSAGE INTERFACE

Overview

An instance of **org.apache.camel.Message** type can represent any kind of message (*In* or *Out*). Figure 11.1, “Message Inheritance Hierarchy” 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 11.1. Message Inheritance Hierarchy



The Message interface

Example 11.1, “Message Interface” shows the definition of the **org.apache.camel.Message** interface.

Example 11.1. Message Interface

```
package org.apache.camel;

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

import javax.activation.DataHandler;

public interface Message {

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

    Exchange getExchange();

    boolean isFault();
    void    setFault(boolean fault);
```

```
    Object getHeader(String name);
    Object getHeader(String name, Object defaultValue);
    <T> T getHeader(String name, Class<T> type);
    <T> T getHeader(String name, Object defaultValue, Class<T> type);
    Map<String, Object> getHeaders();
    void setHeader(String name, Object value);
    void setHeaders(Map<String, Object> headers);
    Object  removeHeader(String name);
    boolean removeHeaders(String pattern);
    boolean hasHeaders();

    Object getBody();
    Object getMandatoryBody() throws InvalidPayloadException;
    <T> T  getBody(Class<T> type);
    <T> T  getMandatoryBody(Class<T> type) throws
InvalidPayloadException;
    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);

    String createExchangeId();
}
```

Message methods

The **`Message`** interface defines the following methods:

- **`setMessageId()`**, **`getMessageId()`**—Getter and setter methods for the message ID. Whether or not you need to use a message ID in your custom component is an implementation detail.

- **`getExchange()`**—Returns a reference to the parent exchange object.

- **`isFault()`**, **`setFault()`**—Getter and setter methods for the fault flag, which indicates whether or not this message is a fault message.

- **`getHeader()`**, **`getHeaders()`**, **`setHeader()`**, **`setHeaders()`**, **`removeHeader()`**, **`hasHeaders()`**—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()`**, **`getMandatoryBody()`**, **`setBody()`**—Getter and setter methods for the message body. The getMandatoryBody() accessor guarantees that the returned body is non-null, otherwise the **`InvalidPayloadException`** exception is thrown.

- **`getAttachment()`**, **`getAttachments()`**, **`getAttachmentNames()`**, **`removeAttachment()`**, **`addAttachment()`**, **`setAttachments()`**, **`hasAttachments()`**—Methods to get, set, add, and remove attachments.

- **`copy()`**—Creates a new, identical (including the message ID) copy of the current custom message object.

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

- **`createExchangeId()`**—Returns the unique ID for this exchange, if the message implementation is capable of providing an ID; otherwise, return **`null`**.

11.2. IMPLEMENTING THE MESSAGE INTERFACE

How to implement a custom message

Example 11.2, “Custom Message Implementation” outlines how to implement a message by extending the **`DefaultMessage`** class.

Example 11.2. Custom Message Implementation

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

1 public class CustomMessage extends DefaultMessage {

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

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

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

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

    @Override
6     protected void populateInitialHeaders(Map<String, Object> map) {
```

```
// Initialize headers from underlying message (lazy creation).
    }

    @Override
    protected void populateInitialAttachments(Map<String, DataHandler>
map) {
            // Initialize attachments from underlying message (lazy
creation).
    }
}
```

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

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

3. Override the **toString()** method to customize message stringification.

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

5. 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.

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

7. The **populateInitialAttachments()** method works in conjunction with the attachment getter and setter methods to implement lazy access to the attachments. This method extracts the message attachments and inserts them into the hash map, **map**. The **populateInitialAttachments()** method is automatically called when a user attempts to access an attachment (or attachments) for the first time by calling **getAttachment()**, **getAttachments()**, **getAttachmentNames()**, or **addAttachment()**.

INDEX

Symbols

@Converter, Implement an annotated converter class

A

AsyncCallback, Asynchronous processing

asynchronous producer
- implementing, How to implement an asynchronous producer

AsyncProcessor, Asynchronous processing

auto-discovery
- configuration, Configuring auto-discovery

C

Component
- createEndpoint(), URI parsing
- definition, The Component interface
- methods, Component methods

component prefix, Component

components, Component
- bean properties, Define bean properties on your component class
- configuring, Installing and configuring the component
- implementation steps, Implementation steps
- installing, Installing and configuring the component
- interfaces to implement, Which interfaces do you need to implement?
- parameter injection, Parameter injection
- Spring configuration, Configure the component in Spring

Consumer, Consumer

consumers, Consumer
- event-driven, Event-driven pattern, Implementation steps
- polling, Polling pattern, Implementation steps
- scheduled, Scheduled poll pattern, Implementation steps
- threading, Overview

D

DefaultComponent
- createEndpoint(), URI parsing

DefaultEndpoint, Event-driven endpoint implementation
- createExchange(), Event-driven endpoint implementation
- createPollingConsumer(), Event-driven endpoint implementation

getCamelConext(), Event-driven endpoint implementation
getComponent(), Event-driven endpoint implementation
getEndpointUri(), Event-driven endpoint implementation

E

Endpoint, Endpoint
createConsumer(), Endpoint methods
createExchange(), Endpoint methods
createPollingConsumer(), Endpoint methods
createProducer(), Endpoint methods
getCamelContext(), Endpoint methods
getEndpointURI(), Endpoint methods
interface definition, The Endpoint interface
isLenientProperties(), Endpoint methods
isSingleton(), Endpoint methods
setCamelContext(), Endpoint methods

endpoint
event-driven, Event-driven endpoint implementation
scheduled, Scheduled poll endpoint implementation

endpoints, Endpoint
Exchange, Exchange, The Exchange interface
copy(), Exchange methods
getExchangeId(), Exchange methods
getIn(), Accessing message headers, Exchange methods
getOut(), Exchange methods
getPattern(), Exchange methods
getProperties(), Exchange methods
getProperty(), Exchange methods
getUnitOfWork(), Exchange methods
removeProperty(), Exchange methods
setExchangeId(), Exchange methods
setIn(), Exchange methods
setOut(), Exchange methods

setProperty(), Exchange methods

setUnitOfWork(), Exchange methods

exchange
in capable, Testing the exchange pattern

out capable, Testing the exchange pattern

exchange properties
accessing, Wrapping the exchange accessors

ExchangeHelper, The ExchangeHelper Class
getContentType(), Get the In message's MIME content type

getMandatoryHeader(), Accessing message headers, Wrapping the exchange accessors

getMandatoryInBody(), Wrapping the exchange accessors

getMandatoryOutBody(), Wrapping the exchange accessors

getMandatoryProperty(), Wrapping the exchange accessors

isInCapable(), Testing the exchange pattern

isOutCapable(), Testing the exchange pattern

resolveEndpoint(), Resolve an endpoint

exchanges, Exchange

I

in message
MIME type, Get the In message's MIME content type

M

Message, Message
getHeader(), Accessing message headers

message headers
accessing, Accessing message headers

messages, Message

P

pipeline, Pipelining model

Processor, Processor interface
implementing, Implementing the Processor interface

producer, Producer

Producer, Producer

- createExchange(), Producer methods
- getEndpoint(), Producer methods
- process(), Producer methods

producers

- asynchronous, Asynchronous producer
- synchronous, Synchronous producer

S

ScheduledPollEndpoint, Scheduled poll endpoint implementation

simple processor

- implementing, Implementing the Processor interface

synchronous producer

- implementing, How to implement a synchronous producer

T

type conversion

- runtime process, Type conversion process

type converter

- annotating the implementation, Implement an annotated converter class
- discovery file, Create a TypeConverter file
- implementation steps, How to implement a type converter
- mater, Master type converter
- packaging, Package the type converter
- slave, Master type converter

TypeConverter, Type converter interface

TypeConverterLoader, Type converter loader

U

useIntrospectionOnEndpoint(), Disabling endpoint parameter injection