talend*
*open integration solutions

Talend Enterprise Mediation
Developer Guide

Covers Apache Camel 2.9.x series

Talend Enterprise Mediation: Developer Guide

Publication date 3 May 2012
Copyright © 2011-2012 Talend Inc.

Copyleft

Notices

Table of Contents

Chapter 1. Introduction

Talend ESB provides a fully supported, stable, production ready distribution of the industry leading open source integration framework Apache Camel. Apache Camel uses well known Enterprise Integration Patterns to make message based system integration simpler yet powerful and scalable.

The Apache Camel uses a lightweight, component based architecture which allows great flexibility in deployment scenarios: as stand-alone JVM applications or embedded in a servlet container such as Tomcat, or within a JEE server, or in an OSGi container such as Equinox.

Apache Camel and Talend ESB come out of the box with an impressive set of available components for all commonly used protocols like http, https, ftp, xmpp, rss and many more. A large number of data formats like EDI, JSON, CSV, HL7 and languages like JS, Python, Scala, are supported out of the box. Its extensible architecture allows developers to easily add support for proprietary protocols and data formats.

The Talend ESB distribution supplements Apache Camel with support for OSGi deployment, support for integrating Talend jobs on Camel routes and a number of advanced examples. Its OSGi container uses Apache Karaf, a lightweight container providing advanced features such as provisioning, hot deployment, logger system, dynamic configuration, complete shell environment, and other features.

Chapter 2. Enterprise Integration Patterns

Camel supports most of the Enterprise Integration Patterns from the excellent book by Gregor Hohpe and Bobby Woolf.

2.1. List of EIPs

2.1.1. Messaging Systems

	Section 2.24, "Message Channel"	How does one application communicate with another using messaging?
	Section 2.22, "Message"	How can two applications connected by a message channel exchange a piece of information?
	Section 2.34, "Pipes and Filters"	How can we perform complex processing on a message while maintaining independence and flexibility?
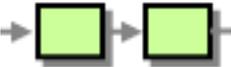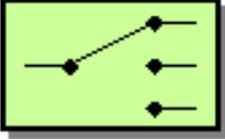
	Section 2.28, "Message Router"	How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?
	Section 2.29, "Message Translator"	How can systems using different data formats communicate with each other using messaging?
	Section 2.26, "Message Endpoint"	How does an application connect to a messaging channel to send and receive messages?
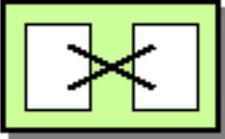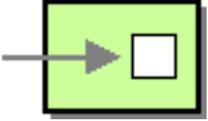
2.1.2. Messaging Channels

	Section 2.35, "Point to Point Channel"	How can the caller be sure that exactly one receiver will receive the document or perform the call?
	Section 2.37, "Publish Subscribe Channel"	How can the sender broadcast an event to all interested receivers?
	Section 2.10, "Dead Letter Channel"	What will the messaging system do with a message it cannot deliver?
	Section 2.17, "Guaranteed Delivery"	How can the sender make sure that a message will be delivered, even if the messaging system fails?
	Section 2.23, "Message Bus"	What is an architecture that enables separate applications to work together, but in a de-coupled fashion such that applications can be easily added or removed without affecting the others?
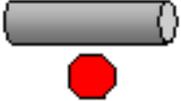
2.1.3. Message Construction
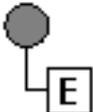
	Section 2.16, "Event Message"	How can messaging be used to transmit events from one application to another?
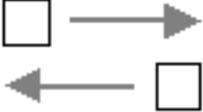
	Section 2.39, "Request Reply"	When an application sends a message, how can it get a response from the receiver?
	Section 2.9, "Correlation Identifier"	How does a requestor that has received a reply know which request this is the reply for?
	Section 2.41, "Return Address"	How does a replier know where to send the reply?
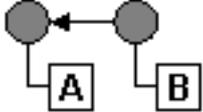
2.1.4. Message Routing
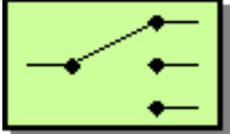
	Section 2.6, "Content Based Router"	How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?
	Section 2.27, "Message Filter"	How can a component avoid receiving uninteresting messages?
	Section 2.14, "Dynamic Router"	How can you avoid the dependency of the router on all possible destinations while maintaining its efficiency?
	Section 2.38, "Recipient List"	How do we route a message to a list of (static or dynamically) specified recipients?
	Section 2.48, "Splitter"	How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?
	Aggregator	How do we combine the results of individual, but related messages so that they can be processed as a whole?
	Section 2.40, "Resequencer"	How can we get a stream of related but out-of-sequence messages back into the correct order?
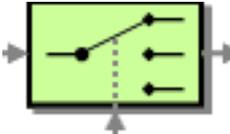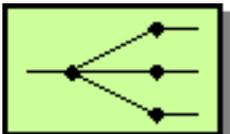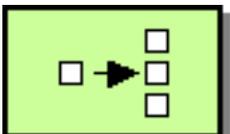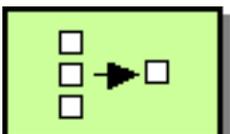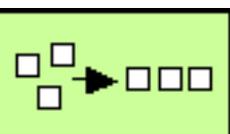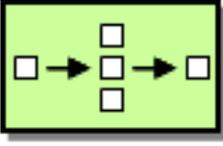
	Section 2.5, "Composed Message Processor"	How can you maintain the overall message flow when processing a message consisting of multiple elements, each of which may require different processing?
	Section 2.44, "Scatter-Gather"	How do you maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply?
	Section 2.42, "Routing Slip"	How do we route a message consecutively through a series of processing steps when the sequence of steps is not known at design-time and may vary for each message?
	Section 2.49, "Throttler"	How can I throttle messages to ensure that a specific endpoint does not get overloaded, or we don't exceed an agreed SLA with some external service?
	Section 2.43, "Sampling"	How can I sample one message out of many in a given period to avoid downstream route does not get overloaded?
	Section 2.11, "Delayer"	How can I delay the sending of a message?
	Section 2.19, "Load Balancer"	How can I balance load across a number of endpoints?
	Section 2.32, "Multicast"	How can I route a message to a number of endpoints at the same time?
	Section 2.21, "Loop"	How can I repeat processing a message in a loop?

2.1.5. Message Transformation
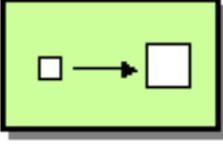
	Section 2.7, "Content Enricher"	How do we communicate with another system if the message originator does not have all the required data items available?
	Section 2.8, "Content Filter"	How do you simplify dealing with a large message, when you are interested only in a few data items?
	Section 2.3, "Claim Check"	How can we reduce the data volume of message sent across the system without sacrificing information content?
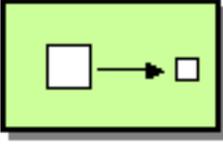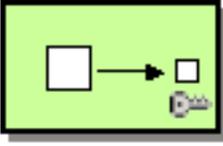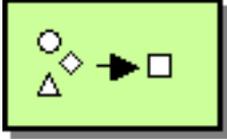
	Section 2.33, "Normalizer"	How do you process messages that are semantically equivalent, but arrive in a different format?
	Section 2.47, "Sort"	How can I sort the body of a message?
	Section 2.51, "Validate"	How can I validate a message?

2.1.6. Messaging Endpoints

	Section 2.31, "Messaging Mapper"	How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?
	Section 2.15, "Event Driven Consumer"	How can an application automatically consume messages as they become available?
	Section 2.36, "Polling Consumer"	How can an application consume a message when the application is ready?
	Section 2.4, "Competing Consumers"	How can a messaging client process multiple messages concurrently?
	Section 2.25, "Message Dispatcher"	How can multiple consumers on a single channel coordinate their message processing?
	Section 2.45, "Selective Consumer"	How can a message consumer select which messages it wishes to receive?
	Section 2.13, "Durable Subscriber"	How can a subscriber avoid missing messages while it's not listening for them?
	Section 2.18, "Idempotent Consumer"	How can a message receiver deal with duplicate messages?
	Section 2.50, "Transactional Client"	How can a client control its transactions with the messaging system?
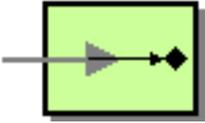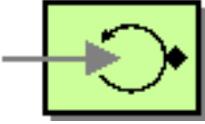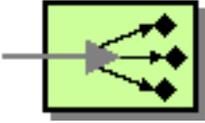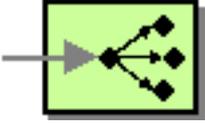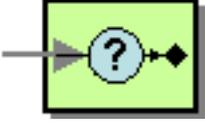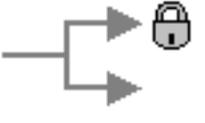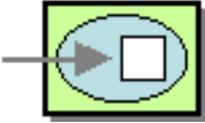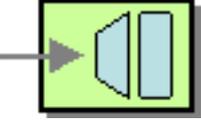
	Section 2.30, "Messaging Gateway"	How do you encapsulate access to the messaging system from the rest of the application?
	Section 2.46, "Service Activator"	How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?
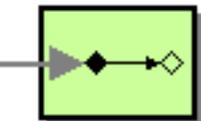
2.1.7. System Management
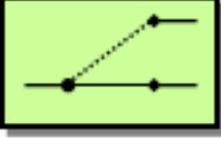
	Section 2.12, "Detour"	How can you route a message through intermediate steps to perform validation, testing or debugging functions?
	Section 2.52, "Wire Tap"	How do you inspect messages that travel on a point-to-point channel?
	Log	How can I log processing a message?
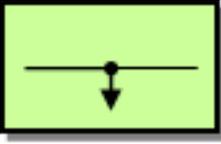
2.2. Aggregator

2.2.1. Aggregator Pattern

The Aggregator from the EIP patterns allows you to combine a number of messages together into a single message.
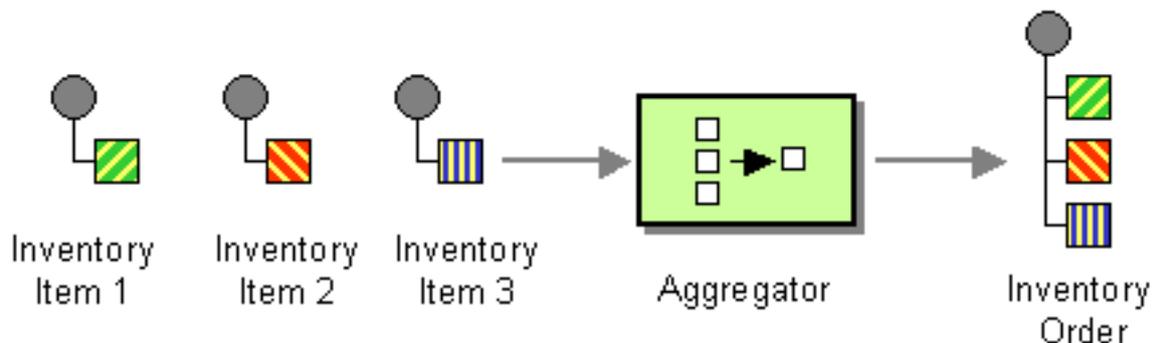
A correlation Expression is used to determine the messages which should be aggregated together. If you want to aggregate all messages into a single message, just use a constant expression. An AggregationStrategy is used to combine all the message exchanges for a single correlation key into a single message exchange.

The aggregator provides a pluggable repository which you can implement your own `org.apache.camel.spi.AggregationRepository`. If you need a persistent repository then you can use either Camel HawtDB or SQL Component.

You can manually complete all current aggregated exchanges by sending in a message containing the header Exchange.AGGREGATION_COMPLETE_ALL_GROUPS set to true. The message is considered a signal message only, the message headers/contents will not be processed otherwise.

The Apache Camel website maintains several examples of this EIP in use.

2.2.2. Aggregator options

The aggregator supports the following options:

Option	Default	Description
correlationExpression		Mandatory Expression which evaluates the correlation key to use for aggregation. The Exchange which has the same correlation key is aggregated together. If the correlation key could not be evaluated an Exception is thrown. You can disable this by using the `ignoreBadCorrelationKeys` option.
aggregationStrategy		Mandatory `AggregationStrategy` which is used to *merge* the incoming Exchange with the existing already merged exchanges. At first call the `oldExchange` parameter is `null`. On subsequent invocations the `oldExchange` contains the merged exchanges and `newExchange` is of course the new incoming Exchange. The strategy can also be a TimeoutAwareAggregationStrategy implementation, supporting the timeout callback. Here, Camel will invoke the `timeout` method when the timeout occurs. Notice that the values for index and total parameters will be -1, and the timeout parameter will only be provided if configured as a fixed value.
strategyRef		A reference to lookup the `AggregationStrategy` in the Registry.
completionSize		number of messages aggregated before the aggregation is complete. This option can be set as either a fixed value or using an Expression which allows you to evaluate a size dynamically; it will use `Integer` as result. If both are set, Camel will fallback to use the fixed value if the Expression result was `null` or `0`.
completionTimeout		Time in milliseconds that an aggregated exchange should be inactive before it is complete. This option can be set as either a fixed value or using an Expression which allows you to evaluate a timeout dynamically; it will use `Long` as result. If both are set Camel will fallback to use the fixed value if the Expression result was `null` or `0`. You cannot use this option together with completionInterval, only one of the two can be used.
completionInterval		A repeating period in milliseconds by which the aggregator will complete all current aggregated exchanges. Camel has a background task which is triggered every period. You cannot use this option together with completionTimeout, only one of them can be used.
completionPredicate		A Predicate to indicate when an aggregated exchange is complete.

Option	Default	Description
completionFromBatchConsumer	`false`	This option is if the exchanges are coming from a Batch Consumer. Then when enabled the Section 2.2, "Aggregator" will use the batch size determined by the Batch Consumer in the message header `CamelBatchSize`. See more details at Batch Consumer. This can be used to aggregate all files consumed from a File endpoint in that given poll.
forceCompletionOnStop	`false`	Indicates completing all current aggregated exchanges when the context is stopped.
eagerCheckCompletion	`false`	Whether or not to eager check for completion when a new incoming Exchange has been received. This option influences the behavior of the `completionPredicate` option as the Exchange being passed in changes accordingly. When `false` the Exchange passed in the Predicate is the *aggregated* Exchange which means any information you may store on the aggregated Exchange from the `AggregationStrategy` is available for the Predicate. When `true` the Exchange passed in the Predicate is the *incoming* Exchange, which means you can access data from the incoming Exchange.
groupExchanges	`false`	If enabled then Camel will group all aggregated Exchanges into a single combined `org.apache.camel.impl.GroupedExchange` holder class that holds all the aggregated Exchanges. And as a result only one Exchange is being sent out from the aggregator. Can be used to combine many incoming Exchanges into a single output Exchange without coding a custom `AggregationStrategy` yourself. Note this option does not support persistant aggregator repositories.
ignoreInvalidCorrelationKeys	`false`	Whether or not to ignore correlation keys which could not be evaluated to a value. By default Camel will throw an Exception, but you can enable this option and ignore the situation instead.
closeCorrelationKeyOnCompletion		Whether or not too *late* Exchanges should be accepted or not. You can enable this to indicate that if a correlation key has already been completed, then any new exchanges with the same correlation key be denied. Camel will then throw a `closedCorrelationKeyException` exception. When using this option you pass in a `integer` which is a number for a LRUCache which keeps that last X number of closed correlation keys. You can pass in 0 or a negative value to indicate a unbounded cache. By passing in a number you are ensured that cache won't grow too big if you use a log of different correlation keys.
discardOnCompletionTimeout	`false`	Whether or not exchanges which complete due to a timeout should be discarded. If enabled then when a timeout occurs the aggregated message will **not** be sent out but dropped (discarded).
aggregationRepository		Allows you to plugin you own implementation of Camel's `AggregationRepository` class which keeps track of the current inflight aggregated exchanges. Camel uses by default a memory based implementation.
aggregationRepositoryRef		Reference to lookup a `aggregationRepository` in the Registry.

Option	Default	Description
parallelProcessing	false	When aggregated are completed they are being send out of the aggregator. This option indicates whether or not Camel should use a thread pool with multiple threads for concurrency. If no custom thread pool has been specified then Camel creates a default pool with 10 concurrent threads.
executorService		If using `parallelProcessing` you can specify a custom thread pool to be used. In fact also if you are not using `parallelProcessing` this custom thread pool is used to send out aggregated exchanges as well.
executorServiceRef		Reference to lookup a `executorService` in the Registry
timeoutCheckerExecutorService		If using either of the completionTimeout, completionTimeoutExpression, or completionInterval options a background thread is created to check for the completion for every aggregator. Set this option to provide a custom thread pool to be used rather than creating a new thread for every aggregator.
timeoutCheckerExecutorServiceRef		Reference to lookup a timeoutCheckerExecutorService in the Registry.

2.2.3. Exchange Properties

The following properties are set on each aggregated Exchange:

header	type	description
CamelAggregatedSize	int	The total number of Exchanges aggregated into this combined Exchange.
CamelAggregatedCompletedBy	String	Indicator how the aggregation was completed as a value of either: `predicate`, `size`, `consumer`, `timeout` or `interval`.

2.2.4. About AggregationStrategy

The `AggregationStrategy` is used for aggregating the old (lookup by its correlation id) and the new exchanges together into a single exchange. Possible implementations include performing some kind of combining or delta processing, such as adding line items together into an invoice or just using the newest exchange and removing old exchanges such as for state tracking or market data prices; where old values are of little use.

Notice the aggregation strategy is a mandatory option and must be provided to the aggregator.

2.2.5. About completion

When aggregation Exchanges at some point you need to indicate that the aggregated exchanges is complete, so they can be send out of the aggregator. Camel allows you to indicate completion in various ways as follows:

• completionTimeout - Is an inactivity timeout in which is triggered if no new exchanges have been aggregated for that particular correlation key within the period.

- completionInterval - Once every X period all the current aggregated exchanges are completed.

- completionSize - Is a number indicating that after X aggregated exchanges it's complete.

- completionPredicate - Runs a Predicate when a new exchange is aggregated to determine if we are complete or not

- completionFromBatchConsumer - Special option for Batch Consumer which allows you to complete when all the messages from the batch has been aggregated. |

- forceCompletionOnStop - Indicates to complete all current aggregated exchanges when the context is stopped.

Notice that all the completion ways are per correlation key. And you can combine them in any way you like. It's basically the first which triggers that wins. So you can use a completion size together with a completion timeout. Only completionTimeout and completionInterval cannot be used at the same time.

Notice the completion is a mandatory option and must be provided to the aggregator. If not provided Camel will thrown an Exception on startup.

2.3. Claim Check

The Claim Check from the EIP patterns allows you to replace message content with a claim check (a unique key), which can be used to retrieve the message content at a later time. The message content is stored temporarily in a persistent store like a database or file system. This pattern is very useful when message content is very large (thus it would be expensive to send around) and not all components require all information.

It can also be useful in situations where you cannot trust the information with an outside party; in this case, you can use the Claim Check to hide the sensitive portions of data.
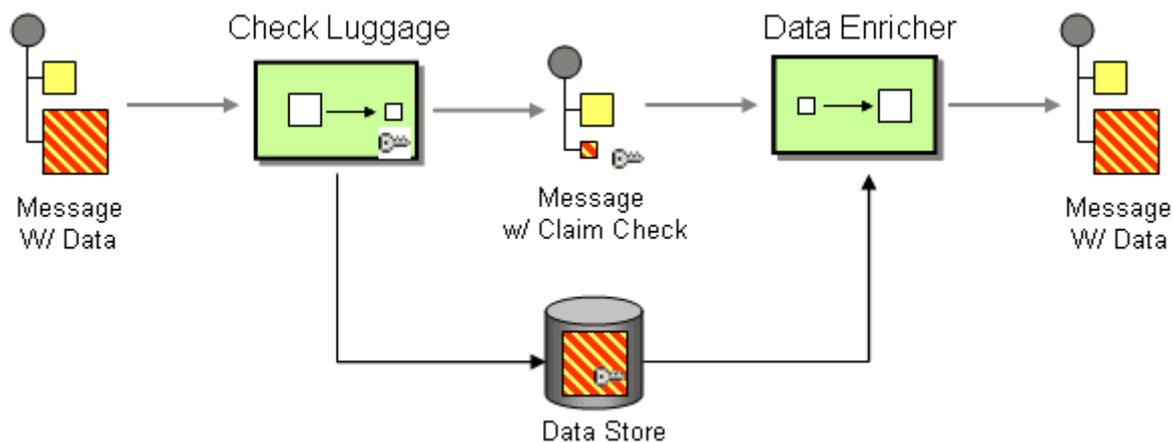
2.3.1. Example

In this example we want to replace a message body with a claim check, and restore the body at a later step.

Using the Fluent Builders

```
from("direct:start").to("bean:checkLuggage", "mock:testCheckpoint", "
    bean:dataEnricher", "mock:result");
```

Using the Spring XML Extensions

```
<route>
    <from uri="direct:start"/>
    <pipeline>
        <to uri="bean:checkLuggage"/>
        <to uri="mock:testCheckpoint"/>
        <to uri="bean:dataEnricher"/>
        <to uri="mock:result"/>
    </pipeline>
</route>
```

The example route is pretty simple - it's a Pipeline. In a real application you would have some other steps where the `mock:testCheckpoint` endpoint is in the example.

The message is first sent to the `checkLuggage` bean which looks like

```
public static final class CheckLuggageBean {
    public void checkLuggage(Exchange exchange, @Body String body,
        @XPath("/order/@custId") String custId) {
        // store the message body into the data store,
        // using the custId as the claim check
        dataStore.put(custId, body);
        // add the claim check as a header
        exchange.getIn().setHeader("claimCheck", custId);
        // remove the body from the message
        exchange.getIn().setBody(null);
    }
}
```

This bean stores the message body into the data store, using the `custId` as the claim check. In this example, we're just using a `HashMap` to store the message body; in a real application you would use a database or file system, etc. Next the claim check is added as a message header for use later. Finally we remove the body from the message and pass it down the pipeline.

The next step in the pipeline is the `mock:testCheckpoint` endpoint which is just used to check that the message body is removed, claim check added, etc.

To add the message body back into the message, we use the `dataEnricher` bean which looks like

```
public static final class DataEnricherBean {
    public void addDataBackIn(Exchange exchange, @Header("claimCheck")
        String claimCheck) {
        // query the data store using the claim check as the key and add the
        // data back into the message body
        exchange.getIn().setBody(dataStore.get(claimCheck));
        // remove the message data from the data store
        dataStore.remove(claimCheck);
        // remove the claim check header
        exchange.getIn().removeHeader("claimCheck");
    }
}
```

This bean queries the data store using the claim check as the key and then adds the data back into the message. The message body is then removed from the data store and finally the claim check is removed. Now the message is back to what we started with!

For full details, check the example source here:

camel-core/src/test/java/org/apache/camel/processor/ClaimCheckTest.java

2.4. Competing Consumers

2.4.1. Competing Consumers

Camel supports the Competing Consumers from the EIP patterns using a few different components.
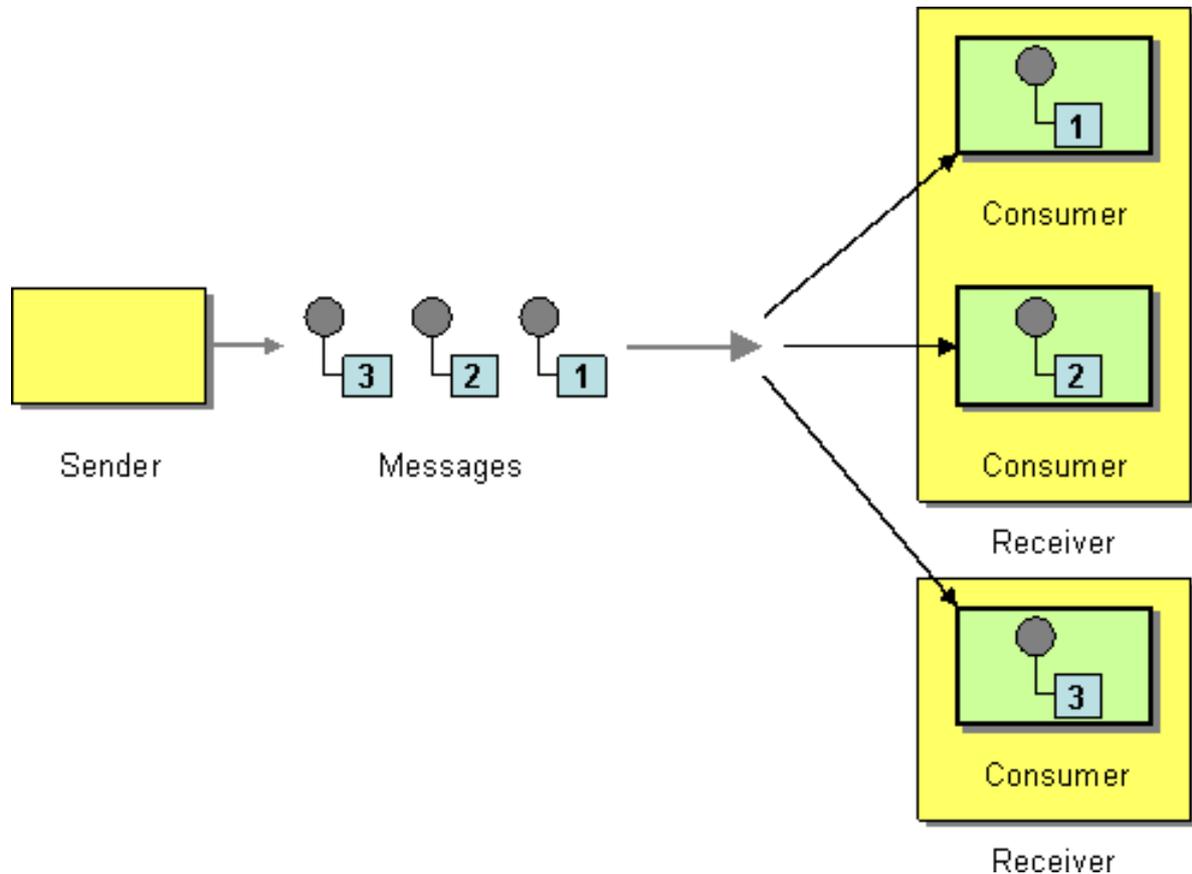
You can use the following components to implement competing consumers:-

- Section 3.39, "SEDA" for SEDA based concurrent processing using a thread pool

- Section 3.25, "JMS" for distributed SEDA based concurrent processing with queues which support reliable load balancing, failover and clustering.

2.4.2. Enabling Competing Consumers with JMS

To enable Competing Consumers you just need to set the **concurrentConsumers** property on the Section 3.25, "JMS" endpoint.

For example

```
from("jms:MyQueue?concurrentConsumers=5").bean(SomeBean.class);
```

Or in Spring DSL:

```
<route>
```

_____

```
  <from uri="jms:MyQueue?concurrentConsumers=5"/>
  <to uri="bean:someBean"/>
</route>
```

Or just run multiple JVMs of any ??? or Section 3.25, "JMS" route.

2.5. Composed Message Processor

The Composed Message Processor from the EIP patterns allows you to process a composite message by splitting it up, routing the sub-messages to appropriate destinations and the re-aggregating the responses back into a single message.
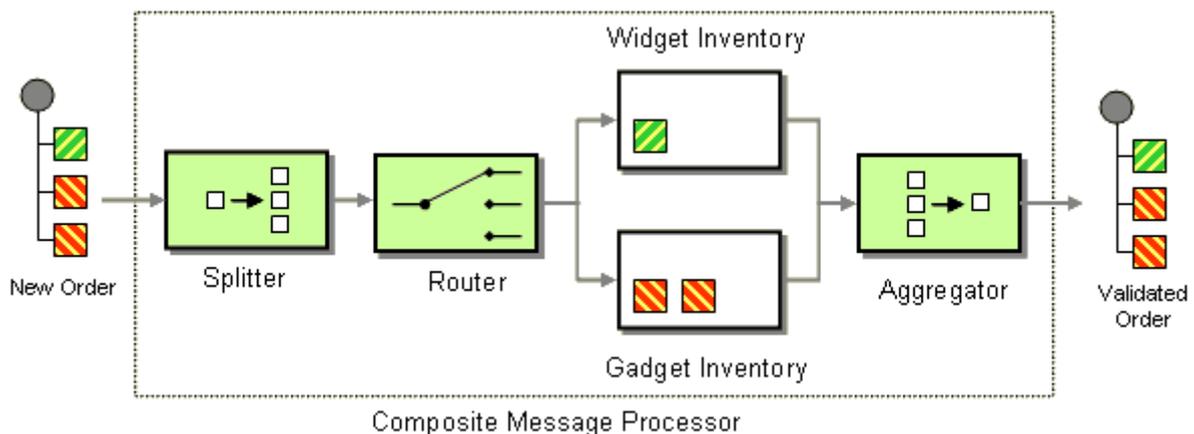
2.5.1. Example

In this example we want to check that a multipart order can be filled. Each part of the order requires a check at a different inventory.

```
// split up the order so individual OrderItems can be
validated by the appropriate bean
from("direct:start")
    .split().body()
    .choice()
        .when().method("orderItemHelper", "isWidget")
            .to("bean:widgetInventory")
        .otherwise()
            .to("bean:gadgetInventory")
    .end()
    .to("seda:aggregate");

// collect and re-assemble the validated OrderItems into an order again
from("seda:aggregate")
    .aggregate(new MyOrderAggregationStrategy()).header("orderId")
.completionTimeout(1000L)
.to("mock:result");
```

Using the Spring XML Extensions:

```
<route>
```

```
  <from uri="direct:start"/>
  <split>
    <simple>body</simple>
    <choice>
      <when>
        <method bean="orderItemHelper" method="isWidget"/>
 <to uri="bean:widgetInventory"/>
      </when>
      <otherwise>
 <to uri="bean:gadgetInventory"/>
      </otherwise>
    </choice>
    <to uri="seda:aggregate"/>
  </split>
</route>

<route>
  <from uri="seda:aggregate"/>
  <aggregate strategyRef="myOrderAggregatorStrategy"
     completionTimeout="1000">
    <correlationExpression>
      <simple>header.orderId</simple>
    </correlationExpression>
    <to uri="mock:result"/>
  </aggregate>
</route>
```

To do this we split up the order using a Section 2.48, "Splitter". The Section 2.48, "Splitter" then sends individual OrderItems to a Section 2.6, "Content Based Router" which checks the item type. Widget items get sent for checking in the widgetInventory bean and gadgets get sent to the gadgetInventory bean. Once these OrderItems have been validated by the appropriate bean, they are sent on to the Section 2.2, "Aggregator" which collects and re-assembles the validated OrderItems into an order again.

When an order is sent it contains a header with the order id. We use this fact when we aggregate, as we configure this .header("orderId") on the aggregate DSL to instruct Camel to use the header with the key orderId as correlation expression.

For full details, check the example source here:

camel-core/src/test/java/org/apache/camel/processor/ComposedMessageProcessorTest.java

2.6. Content Based Router

The Content Based Router from the EIP patterns allows you to route messages to the correct destination based on the contents of the message exchanges.
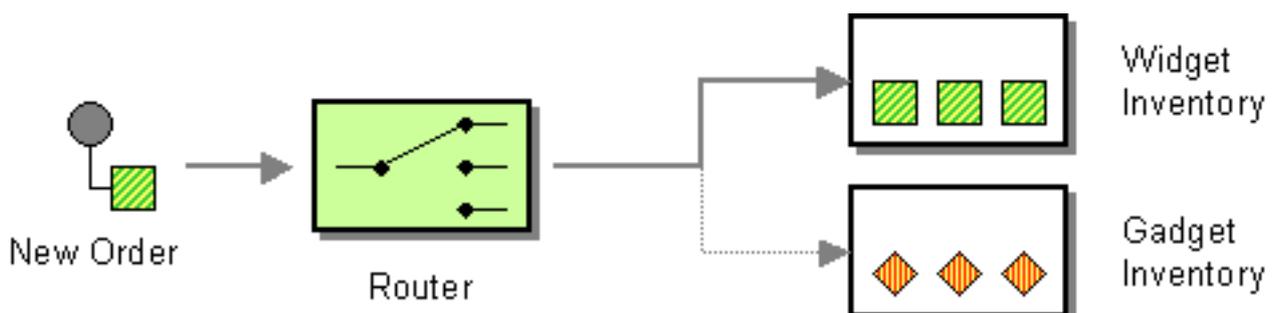
The following example shows how to route a request from an input **seda:a** endpoint to either **seda:b**, **seda:c** or **seda:d** depending on the evaluation of various Predicate expressions

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("mock:error"));

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

Using the Spring XML Extensions

```
<camelContext errorHandlerRef="errorHandler"
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <choice>
            <when>
                <xpath>$foo = 'bar'</xpath>
                <to uri="seda:b"/>
            </when>
            <when>
                <xpath>$foo = 'cheese'</xpath>
                <to uri="seda:c"/>
            </when>
            <otherwise>
                <to uri="seda:d"/>
            </otherwise>
        </choice>
    </route>
</camelContext>
```

For further examples of this pattern in use you could look at the junit test case

2.7. Content Enricher

Camel supports the Content Enricher from the EIP patterns using a Section 2.29, "Message Translator", an arbitrary Processor in the routing logic or using the enrich [18] DSL element to enrich the message.

Enricher

Basic Message

Resource

Enriched Message
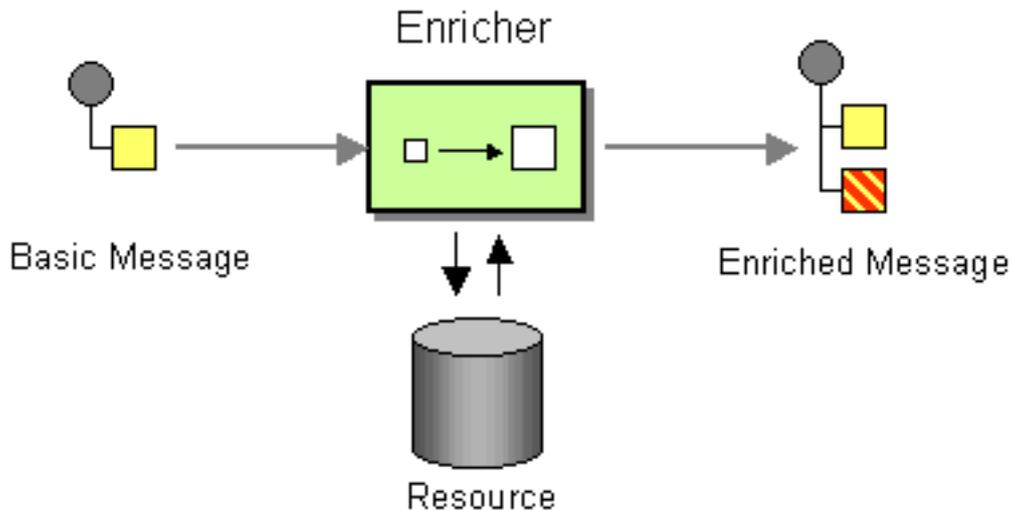
2.7.1. Content enrichment using a Message Translator or a Processor

Using the Fluent Builders

You can use Templating to consume a message from one destination, transform it with something like Section 3.51, "Velocity" or XQuery and then send it on to another destination. For example using InOnly (one way messaging)

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

If you want to use InOut (request-reply) semantics to process requests on the **My.Queue** queue on ??? with a template generated response, then sending responses back to the JMSReplyTo Destination you could use this:

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

We can also use Bean Integration to use any Java method on any bean to act as the transformer

```
from("activemq:My.Queue").
  beanRef("myBeanName", "myMethodName").
  to("activemq:Another.Queue");
```

For further examples of this pattern in use you could look at one of the JUnit tests

- TransformTest

- TransformViaDSLTest

Using Spring XML

```
<route>
  <from uri="activemq:Input"/>
  <bean ref="myBeanName" method="doTransform"/>
  <to uri="activemq:Output"/>
</route>
```

2.7.2. Content enrichment using the enrich DSL element

Camel comes with two flavors of content enricher in the DSL

• `enrich`

• `pollEnrich`

`enrich` is using a `Producer` to obtain the additional data. It is usually used for Section 2.39, "Request Reply" messaging, for instance to invoke an external web service. `pollEnrich` on the other hand is using a Section 2.36, "Polling Consumer" to obtain the additional data. It is usually used for Section 2.16, "Event Message" messaging, for instance to read a file or download a FTP file. Enrich options:

Name	Default Value	Description
`uri`		The endpoint uri for the external service to enrich from. You must use either uri or ref.
`ref`		Refers to the endpoint for the external service to enrich from. You must use either uri or ref.
`strategyRef`		Refers to an AggregationStrategy to be used to merge the reply from the external service, into a single outgoing message. By default Camel will use the reply from the external service as outgoing message.

Using the Fluent Builders

```
AggregationStrategy aggregationStrategy = ...

from("direct:start")
.enrich("direct:resource", aggregationStrategy)
.to("direct:result");

from("direct:resource")
...
```

The content enricher (`enrich`) retrieves additional data from a *resource endpoint* in order to enrich an incoming message (contained in the *original exchange*). An aggregation strategy is used to combine the original exchange and the *resource exchange*. The first parameter of the `AggregationStrategy.aggregate(Exchange, Exchange)` method corresponds to the the original exchange, the second parameter the resource exchange. The results from the resource endpoint are stored in the resource exchange's out-message. Here's an example template for implementing an aggregation strategy.

```
public class ExampleAggregationStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange original, Exchange resource) {
        Object originalBody = original.getIn().getBody();
        Object resourceResponse = resource.getOut().getBody();
        Object mergeResult = ... // combine original body and resourceResponse
        if (original.getPattern().isOutCapable()) {
            original.getOut().setBody(mergeResult);
        } else {
            original.getIn().setBody(mergeResult);
        }
```

```
        return original;
    }

}
```

Using this template the original exchange can be of any pattern. The resource exchange created by the enricher is always an in-out exchange.

Using Spring XML

The same example in the Spring DSL

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <enrich uri="direct:resource" strategyRef="aggregationStrategy"/>
    <to uri="direct:result"/>
  </route>
  <route>
    <from uri="direct:resource"/>
    ...
  </route>
</camelContext>

<bean id="aggregationStrategy" class="..." />
```

2.7.3. Aggregation strategy is optional

The aggregation strategy is optional. If you do not provide it Camel will by default just use the body obtained from the resource.

```
from("direct:start")
  .enrich("direct:resource")
  .to("direct:result");
```

In the route above the message send to the `direct:result` endpoint will contain the output from the `direct:resource` as we do not use any custom aggregation.

And in Spring DSL just omit the `strategyRef` attribute:

```
<route>
    <from uri="direct:start"/>
    <enrich uri="direct:resource"/>
    <to uri="direct:result"/>
</route>
```

2.7.4. Content enrichment using pollEnrich

The `pollEnrich` works just as the `enrich` option however as it uses a Section 2.36, "Polling Consumer" we have 3 methods when polling

• receive

• receiveNoWait

• receive(timeout)

By default Camel will use the `receiveNoWait`. If there is no data then the `newExchange` in the aggregation strategy is `null`.

The same configuration options above for enrich also hold for pollEnrich, but there is also a `timeout` value (in milliseconds) that determines which method will be used:

• timeout is -1 or other negative number then `receive` is selected

• timeout is 0 then `receiveNoWait` is selected

• otherwise `receive(timeout)` is selected

Data from current Exchange not used

`pollEnrich` does **not** access any data from the current Exchange which means when polling it cannot use any of the existing headers you may have set on the Exchange. For example you cannot set a filename in the `Exchange.FILE_NAME` header and use `pollEnrich` to consume only that file. For that you **must** set the filename in the endpoint URI.

In this example we enrich the message by loading the content from the file named inbox/data.txt.

```
from("direct:start")
.pollEnrich("file:inbox?fileName=data.txt")
.to("direct:result");
```

And in XML DSL you do:

```
<route>
<from uri="direct:start"/>
<pollEnrich uri="file:inbox?fileName=data.txt"/>
<to uri="direct:result"/>
</route>
```

If there is no file then the message is empty. We can use a timeout to either wait (potentially forever) until a file exists, or use a timeout to wait a certain period. For example to wait up to 5 seconds you can do:

```
<route>
<from uri="direct:start"/>
<pollEnrich uri="file:inbox?fileName=data.txt" timeout="5000"/>
<to uri="direct:result"/>
</route>
```

2.8. Content Filter

Camel supports the Content Filter from the EIP patterns using one of the following mechanisms in the routing logic to transform content from the inbound message.

• Section 2.29, "Message Translator"

• invoking a Java bean

• Processor object
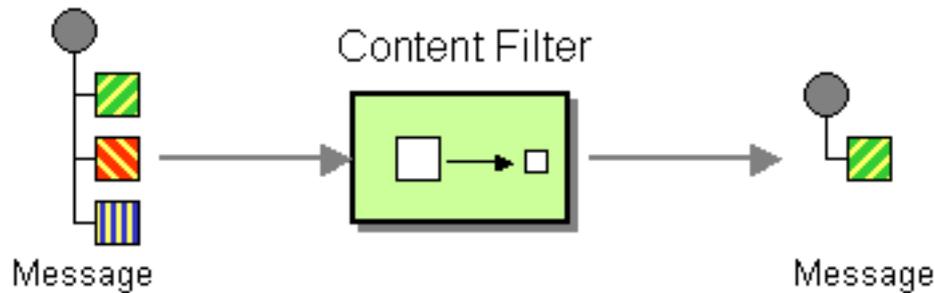
A common way to filter messages is to use an Expression in the DSL like XQuery, SQL or one of the supported Scripting Languages.

Using the Fluent Builders

Here is a simple example using the DSL directly

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

In this example we add our own Processor

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

For further examples of this pattern in use you could look at one of the JUnit tests

- TransformTest

- TransformViaDSLTest

Using Spring XML

```
<route>
  <from uri="activemq:Input"/>
  <bean ref="myBeanName" method="doTransform"/>
  <to uri="activemq:Output"/>
</route>
```

You can also use XPath to filter out part of the message you are interested in:

```
<route>
  <from uri="activemq:Input"/>
  <setBody>
     <xpath resultType="org.w3c.dom.Document">//foo:bar</xpath>
  </setBody>
  <to uri="activemq:Output"/>
</route>
```

2.9. Correlation Identifier

Camel supports the Correlation Identifier from the EIP patterns by getting or setting a header on a Section 2.22, "Message".

When working with the ??? or Section 3.25, "JMS" components the correlation identifier header is called **JMSCorrelationID**. You can add your own correlation identifier to any message exchange to help correlate messages together to a single conversation (or business process).
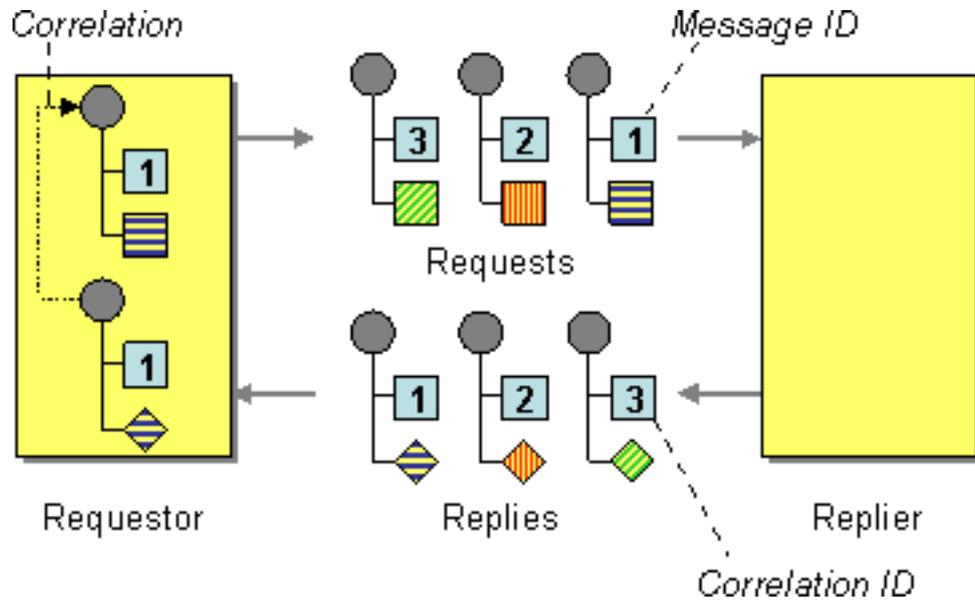
The use of a Correlation Identifier is key to working with the Camel Business Activity Monitoring Framework and can also be highly useful when testing with simulation or canned data such as with the Mock testing framework

2.10. Dead Letter Channel

Camel supports the Dead Letter Channel from the EIP patterns using the DeadLetterChannel processor which is an Error Handler.
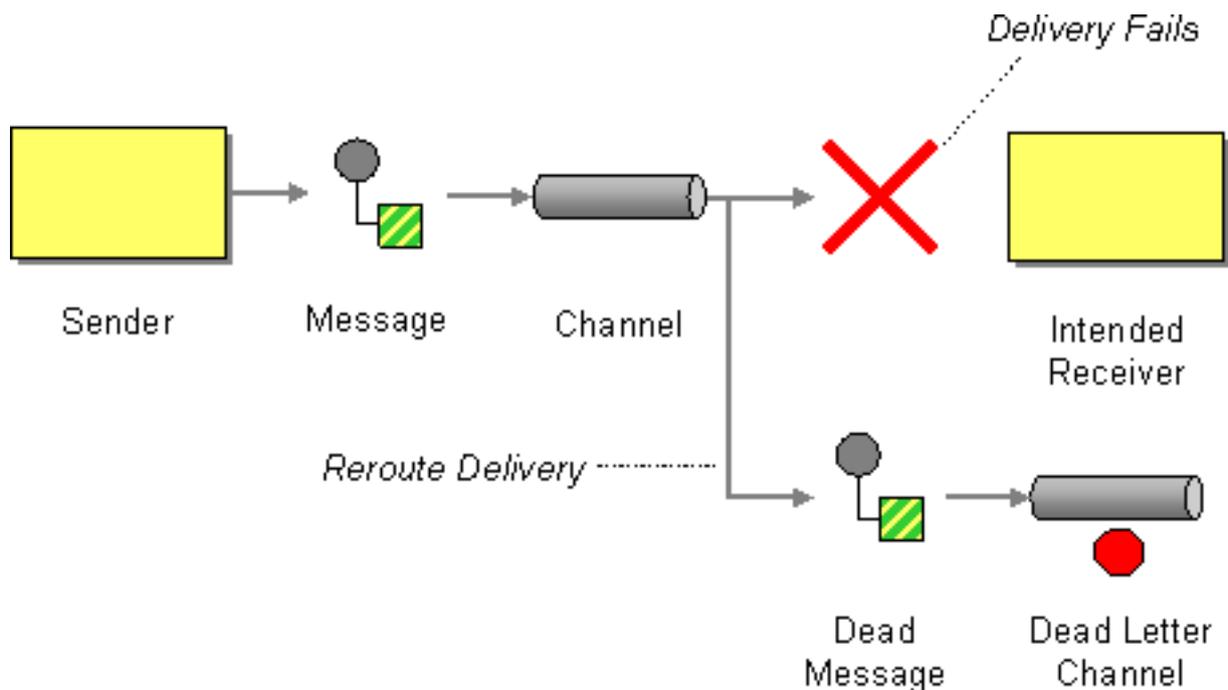
> **Difference between Dead Letter Channel and Default Error Handler**
>
> The major difference is that Section 2.10, "Dead Letter Channel" has a dead letter queue that whenever an Exchange could not be processed is moved to. It will **always** moved failed exchanges to this queue.
>
> Unlike the Default Error Handler that does **not** have a dead letter queue. So whenever an Exchange could not be processed the error is propagated back to the client.
>
> **Notice:** You can adjust this behavior of whether the client should be notified or not with the **handled** option.

2.10.1. Redelivery

It is common for a temporary outage or database deadlock to cause a message to fail to process; but the chances are if it is tried a few more times with some time delay then it will complete fine. So we typically wish to use some kind of redelivery policy to decide how many times to try redeliver a message and how long to wait before redelivery attempts.

The RedeliveryPolicy defines how the message is to be redelivered. You can customize things like

• how many times a message is attempted to be redelivered before it is considered a failure and sent to the dead letter channel

• the initial redelivery timeout

• whether or not exponential backoff is used (i.e. the time between retries increases using a backoff multiplier)

• whether to use collision avoidance to add some randomness to the timings

• delay pattern, see below for details.

Once all attempts at redelivering the message fails then the message is forwarded to the dead letter queue.

2.10.2. About moving Exchange to dead letter queue and using handled

When all attempts of redelivery have failed the Exchange is moved to the dead letter queue (the dead letter endpoint). The exchange is then complete and from the client point of view it was processed. As such the Section 2.10, "Dead Letter Channel" have handled the Exchange.

For instance configuring the dead letter channel, using the fluent builders:

```
errorHandler(deadLetterChannel("jms:queue:dead")
    .maximumRedeliveries(3).redeliverDelay(5000));
```

Using Spring XML Extensions:

```
<route errorHandlerRef="myDeadLetterErrorHandler">
...
</route>

<bean id="myDeadLetterErrorHandler"
```

```
      class="org.apache.camel.builder.DeadLetterChannelBuilder">
<property name="deadLetterUri" value="jms:queue:dead"/>
<property name="redeliveryPolicy" ref="myRedeliveryPolicyConfig"/>
</bean>

<bean id="myRedeliveryPolicyConfig"
      class="org.apache.camel.processor.RedeliveryPolicy">
<property name="maximumRedeliveries" value="3"/>
<property name="redeliveryDelay" value="5000"/>
</bean>
```

The Section 2.10, "Dead Letter Channel" above will clear the caused exception setException(null), by moving the caused exception to a property on the Exchange, with the key Exchange.EXCEPTION_CAUGHT. Then the exchange is moved to the jms:queue:dead destination and the client will not notice the failure.

2.10.3. About moving Exchange to dead letter queue and using the original message

The option **useOriginalMessage** is used for routing the original input message instead of the current message that potentially is modified during routing.

For instance if you have this route:

```
from("jms:queue:order:input")
      .to("bean:validateOrder")
      .to("bean:transformOrder")
      .to("bean:handleOrder");
```

The route listen for JMS messages and validates, transforms and handle it. During this the Exchange payload is transformed/modified. So in case something goes wrong and we want to move the message to another JMS destination, then we can configure our Section 2.10, "Dead Letter Channel" with the **useOriginalBody** option. But when we move the Exchange to this destination we do not know in which state the message is in. Did the error happen in before the transformOrder or after? So to be sure we want to move the original input message we received from jms:queue:order:input. So we can do this by enabling the **useOriginalMessage** option as shown below:

```
// will use original body
   errorHandler(deadLetterChannel("jms:queue:dead")
      .useOriginalMessage().mamimumRedeliveries(5).redeliverDelay(5000);
```

Then the messages routed to the jms:queue:dead is the original input. If we want to manually retry we can move the JMS message from the failed to the input queue, with no problem as the message is the same as the original we received.

2.10.4. OnRedelivery

When Section 2.10, "Dead Letter Channel" is doing redeliver it is possible to configure a Processor that is executed just **before** every redelivery attempt. This can be used for the situations where you need to alter the message before it is redelivered. See below for sample.

We also have support for per **onException** to set a **onRedeliver**. That means you can do special on redelivery for different exceptions, as opposed to onRedelivery set on Section 2.10, "Dead Letter Channel" can be viewed as a global scope.

2.10.5. Redelivery default values

Redelivery is disabled by default. The default redeliver policy uses the following values:

• maximumRedeliveries=0

• redeliverDelay=1000L (1 second)

 • use initialRedeliveryDelay for previous versions

• maximumRedeliveryDelay = 60 * 1000L (60 seconds)

• And the exponential backoff and collision avoidance is turned off.

• The retriesExhaustedLogLevel are set to LoggingLevel.ERROR

• The retryAttemptedLogLevel are set to LoggingLevel.DEBUG

• Stack traces is logged for exhausted messages.

• Handled exceptions is not logged

The maximum redeliver delay ensures that a delay is never longer than the value, default 1 minute. This can happen if you turn on the exponential backoff.

The maximum redeliveries is the number of **re** delivery attempts. By default Camel will try to process the exchange 1 + 5 times. 1 time for the normal attempt and then 5 attempts as redeliveries. Setting the maximumRedeliveries to a negative value such as -1 will then always redelivery (unlimited). Setting the maximumRedeliveries to 0 will disable any re delivery attempt.

Camel will log delivery failures at the DEBUG logging level by default. You can change this by specifying retriesExhaustedLogLevel and/or retryAttemptedLogLevel. See ExceptionBuilderWithRetryLoggingLevelSetTest for an example.

You can turn logging of stack traces on/off. If turned off Camel will still log the redelivery attempt; but it's much less verbose.

2.10.5.1. Redeliver Delay Pattern

Delay pattern is used as a single option to set a range pattern for delays. If used then the following options does not apply: (delay, backOffMultiplier, useExponentialBackOff, useCollisionAvoidance, maximumRedeliveryDelay).

The idea is to set groups of ranges using the following syntax: `limit:delay;limit 2:delay 2;limit 3:delay 3;...;limit N:delay N`

Each group has two values separated with colon

• limit = upper limit

• delay = delay in milliseconds And the groups is again separated with semi colon. The rule of thumb is that the next groups should have a higher limit than the previous group.

Let's clarify this with an example: `delayPattern=5:1000;10:5000;20:20000`

That gives us 3 groups:

• 5:1000

• 10:5000

• 20:20000

___

Resulting in these delays for redelivery attempt:

- Redelivery attempt number 1..4 = 0 ms (as the first group start with 5)

- Redelivery attempt number 5..9 = 1000 ms (the first group)

- Redelivery attempt number 10..19 = 5000 ms (the second group)

- Redelivery attempt number 20.. = 20000 ms (the last group)

Note: The first redelivery attempt is 1, so the first group should start with 1 or higher.

You can start a group with limit 1 to eg have a starting delay: `delayPattern=1:1000;5:5000`

- Redelivery attempt number 1..4 = 1000 ms (the first group)

- Redelivery attempt number 5.. = 5000 ms (the last group)

There is no requirement that the next delay should be higher than the previous. You can use any delay value you like. For example with `delayPattern=1:5000;3:1000` we start with 5 sec delay and then later reduce that to 1 second.

2.10.6. Redelivery header

When a message is redelivered the DeadLetterChannel will append a customizable header to the message to indicate how many times it has been redelivered. The header **CamelRedeliveryMaxCounter**, which is also defined on the `Exchange.REDELIVERY_MAX_COUNTER`, contains the maximum redelivery setting. This header is absent if you use `retryWhile` or have unlimited maximum redelivery configured.

And a boolean flag whether it is being redelivered or not (first attempt). The header **CamelRedelivered** contains a boolean if the message is redelivered or not, which is also defined on the `Exchange.REDELIVERED`.

There's an additional header, `CamelRedeliveryDelay`, to show any dynamically calculated delay from the exchange. This is also defined on the Exchange.REDELIVERY_DELAY. If this header is absent, normal redelivery rules will apply.

2.10.6.1. Which endpoint failed

When Camel routes messages it will decorate the Exchange with a property that contains the **last** endpoint Camel send the Exchange to:

```
String lastEndpointUri = exchange.getProperty(Exchange.TO_ENDPOINT,
    String.class);
```

The `Exchange.TO_ENDPOINT` have the constant value `CamelToEndpoint`.

This information is updated when Camel sends a message to any endpoint. So if it exists it's the **last** endpoint which Camel send the Exchange to.

When for example processing the Exchange at a given Endpoint and the message is to be moved into the dead letter queue, then Camel also decorates the Exchange with another property that contains that **last** endpoint:

```
String failedEndpointUri = exchange.getProperty(Exchange.FAILURE_ENDPOINT,
    String.class);
```

The `Exchange.FAILURE_ENDPOINT` have the constant value `CamelFailureEndpoint`.

This allows for example you to fetch this information in your dead letter queue and use that for error reporting. This is useable if the Camel route is a bit dynamic such as the dynamic Section 2.38, "Recipient List" so you know which endpoints failed.

Notice: These information is kept on the Exchange even if the message was successfully processed by a given endpoint, and then later fails for example in a local Section 3.3, "Bean" processing instead. So beware that this is a hint that helps pinpoint errors.

```
from("activemq:queue:foo")
    .to("http://someserver/somepath")
    .beanRef("foo");
```

Now suppose the route above and a failure happens in the `foo` bean. Then the `Exchange.TO_ENDPOINT` and `Exchange.FAILURE_ENDPOINT` will still contain the value of `http://someserver/somepath` .

2.10.7. Samples

The following example shows how to configure the Dead Letter Channel configuration using the DSL

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        // using dead letter channel with a seda queue for errors
        errorHandler(deadLetterChannel("seda:errors"));

        // here is our route
        from("seda:a").to("seda:b");
    }
};
```

You can also configure the RedeliveryPolicy as this example shows

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        // configures dead letter channel to use seda queue for errors and
        // uses at most 2 redeliveries
        // and exponential backoff
        errorHandler(deadLetterChannel("seda:errors").maximumRedeliveries(2).
            useExponentialBackOff());

        // here is our route
        from("seda:a").to("seda:b");
    }
};
```

2.10.8. How can I modify the Exchange before redelivery?

When Section 2.10, "Dead Letter Channel" is doing redeliver it is possible to configure a Processor that is executed just **before** every redelivery attempt. This can be used for the situations where you need to alter the message before it is redelivered.

Here we configure the Section 2.10, "Dead Letter Channel" to use our processor `MyRedeliveryProcessor` to be executed before each redelivery.

```
// we configure our Dead Letter Channel to invoke
// MyRedeliveryProcessor before a redelivery is
// attempted. This allows us to alter the message before
errorHandler(deadLetterChannel("mock:error").maximumRedeliveries(5)
        .onRedelivery(new MyRedeliverProcessor())
```

```
        // setting delay to zero is just to make unit testing faster
        .redeliveryDelay(0L));
```

And this is the processor `MyRedeliveryProcessor` where we alter the message.

```
// This is our processor that is executed before every redelivery attempt
// here we can do what we want in the Java code, such as altering the message
public class MyRedeliverProcessor implements Processor {

    public void process(Exchange exchange) throws Exception {
        // the message is being redelivered so we can alter it

        // we just append the redelivery counter to the body
        // you can of course do all kind of stuff instead
        String body = exchange.getIn().getBody(String.class);
        int count = exchange.getIn().getHeader(Exchange.REDELIVERY_COUNTER,
            Integer.class);

        exchange.getIn().setBody(body + count);

        // the maximum redelivery was set to 5
        int max = exchange.getIn().getHeader(Exchange.REDELIVERY_MAX_COUNTER,
            Integer.class);
        assertEquals(5, max);
    }
}
```

2.11. Delayer

2.11.1. Delayer

The Delayer Pattern allows you to delay the delivery of messages to some destination. Note: the specified expression is a value in milliseconds to wait from the current time, so if you want to wait 3 sec from now, the expression should be `3000`. You can also use a long value for a fixed value to indicate the delay in milliseconds. See the Spring DSL samples below for Delayer.

Name	Default Value	Description
asyncDelayed	false	If enabled then delayed messages happens asynchronously using a scheduled thread pool.
executorServiceRef		Refers to a custom Thread Pool to be used if asyncDelay has been enabled.
callerRunsWhenRejected	true	Is used if asyncDelayed was enabled. This controls if the caller thread should execute the task if the thread pool rejected the task.

Using the Fluent Builders

```
from("seda:b").delay(1000).to("mock:result");
```

So the above example will delay all messages received on **seda:b** 1 second before sending them to **mock:result**.

You can of course use many different Expression languages such as XPath, XQuery, SQL or various Scripting Languages. You can just delay things a fixed amount of time from the point at which the delayer receives the message. For example to delay things 2 seconds:

```
delayer(2000)
```

The above assumes that the delivery order is maintained and that the messages are delivered in delay order. If you want to reorder the messages based on delivery time, you can use the Section 2.40, "Resequencer" with this pattern. For example:

```
from("activemq:someQueue").resequencer(header("MyDeliveryTime")).
    delay("MyRedeliveryTime").to("activemq:aDelayedQueue");
```

2.11.1.1. Spring DSL

The sample below demonstrates the delay in Spring DSL:

```xml
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <delay>
            <header>MyDelay</header>
        </delay>
        <to uri="mock:result"/>
    </route>
    <route>
        <from uri="seda:b"/>
        <delay>
            <constant>1000</constant>
        </delay>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

2.11.2. Asynchronous delaying

You can let the Section 2.11, "Delayer" use non blocking asynchronous delaying, which means Camel will use a scheduler to schedule a task to be executed in the future. The task will then continue routing. This allows the caller thread to not block and be able to service other messages etc.

2.11.2.1. From Java DSL

You use the asyncDelayed() to enable the async behavior.

```
from("activemq:queue:foo").delay(1000).asyncDelayed().
    to("activemq:aDelayedQueue");
```

2.11.2.2. From Spring XML

You use the asyncDelayed="true" attribute to enable the async behavior.

```xml
<route>
   <from uri="activemq:queue:foo"/>
   <delay asyncDelayed="true">
        <constant>1000</constant>
```

```
    </delay>
    <to uri="activemq:aDealyedQueue"/>
</route>
```

2.11.3. Creating a custom delay

You can use an expression to determine when to send a message using something like this

```
from("activemq:foo").
  delay().method("someBean", "computeDelay").
  to("activemq:bar");
```

then the bean would look like this:

```
public class SomeBean {
  public long computeDelay() {
     long delay = 0;
     // use Java code to compute a delay value in milliseconds
     return delay;
  }
}
```

2.12. Detour

The Detour from the EIP patterns allows you to send messages through additional steps if a control condition is met. It can be useful for turning on extra validation, testing, debugging code when needed.
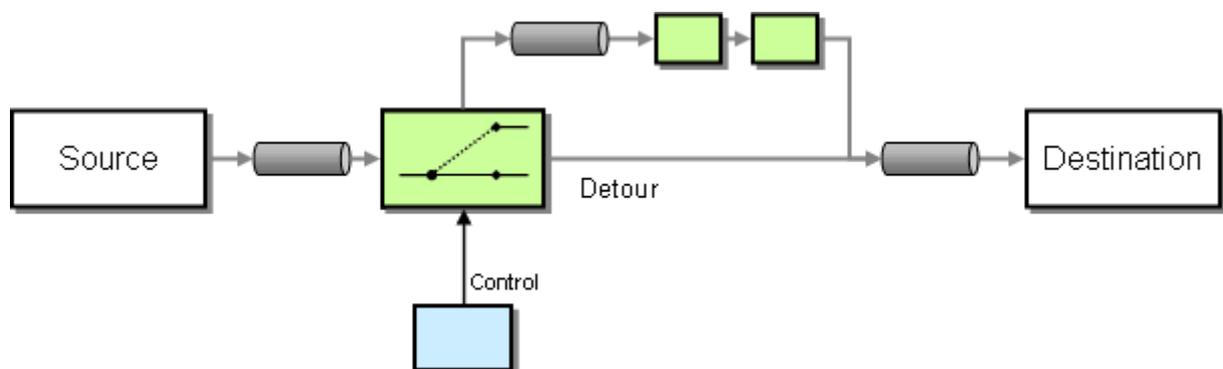
2.12.1. Example

In this example we essentially have a route like `from("direct:start").to("mock:result")` with a conditional detour to the `mock:detour` endpoint in the middle of the route:

```
from("direct:start").choice()
    .when().method("controlBean", "isDetour").to("mock:detour").end()
    .to("mock:result");
```

Using the Spring XML Extensions

```
<route>
```

```
  <from uri="direct:start"/>
    <choice>
      <when>
        <method bean="controlBean" method="isDetour"/>
 <to uri="mock:detour"/>
      </when>
    </choice>
    <to uri="mock:result"/>
  </split>
</route>
```

whether the detour is turned on or off is decided by the `ControlBean`. So, when the detour is on the message is routed to `mock:detour` and then `mock:result`. When the detour is off, the message is routed to `mock:result`.

For full details, check the example source here:

camel-core/src/test/java/org/apache/camel/processor/DetourTest.java

2.13. Durable Subscriber

Camel supports the Durable Subscriber from the EIP patterns using the Section 3.25, "JMS" component which supports publish & subscribe using Topics with support for non-durable and durable subscribers.
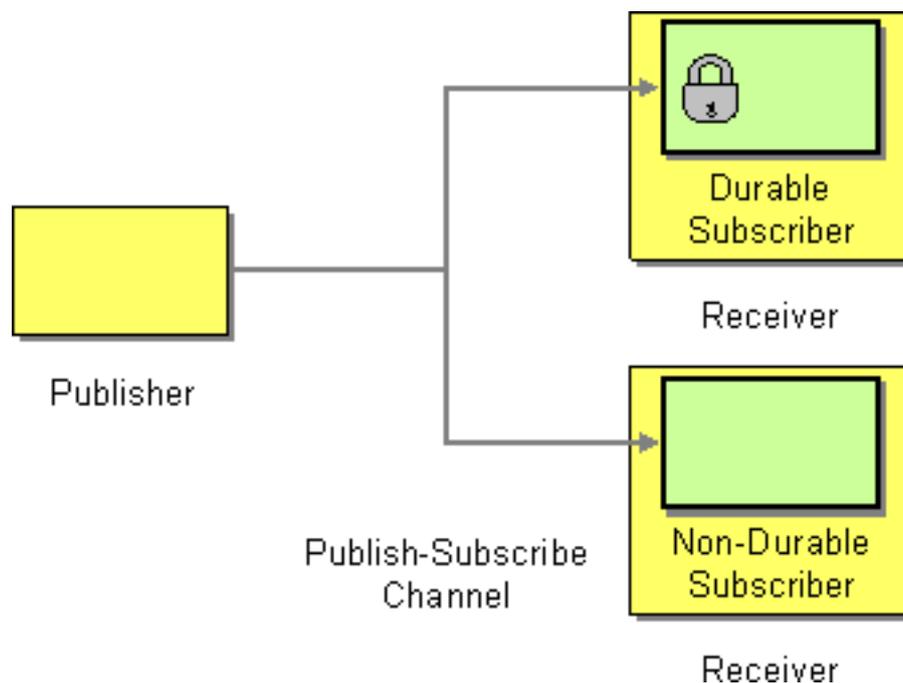
Another alternative is to combine the Section 2.25, "Message Dispatcher" or Section 2.6, "Content Based Router" with ??? or Section 3.27, "JPA" components for durable subscribers then Seda for non-durable.

Here are some examples of creating durable subscribers to a JMS topic. Using the Fluent Builders:

```
from("direct:start").to("activemq:topic:foo");
from("activemq:topic:foo?clientId=1&durableSubscriptionName=bar1").
   to("mock:result1");
from("activemq:topic:foo?clientId=2&durableSubscriptionName=bar2").
   to("mock:result2");
```

Using the Spring XML Extensions:

```
<route>
    <from uri="direct:start"/>
    <to uri="activemq:topic:foo"/>
</route>

<route>
    <from uri="activemq:topic:foo?clientId=1& ...
        durableSubscriptionName=bar1"/>
    <to uri="mock:result1"/>
</route>

<route>
    <from uri="activemq:topic:foo?clientId=2& ...
        durableSubscriptionName=bar2"/>
    <to uri="mock:result2"/>
</route>
```

2.14. Dynamic Router

The Dynamic Router from the EIP patterns allows you to route messages while avoiding the dependency of the router on all possible destinations while maintaining its efficiency.
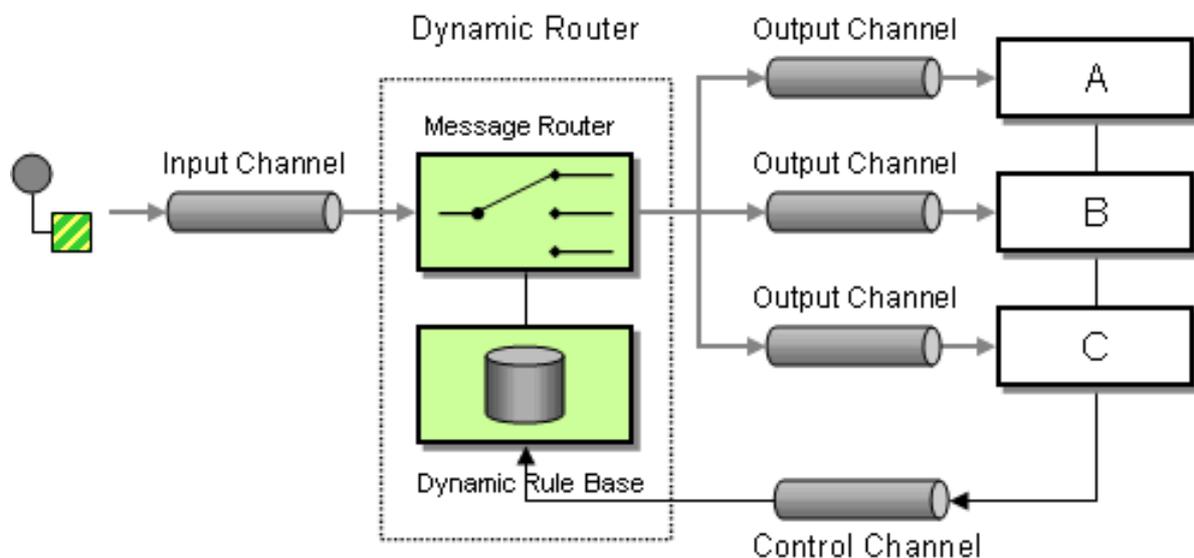
In **Camel 2.5** we introduced a `dynamicRouter` in the DSL which is like a dynamic Section 2.42, "Routing Slip" which evaluates the slip *on-the-fly*.

Beware

You must ensure the expression used for the `dynamicRouter` such as a bean, will return `null` to indicate the end. Otherwise the `dynamicRouter` will keep repeating endlessly.

Option	Default	Description
uriDelimiter	,	Delimiter used if the Expression returned multiple endpoints.

Option	Default	Description
ignoreInvalidEndpoints	`false`	If an endpoint URI could not be resolved, whether it should it be ignored. Otherwise Camel will throw an exception stating that the endpoint URI is not valid.

2.14.1. Dynamic Router

The Section 2.14, "Dynamic Router" will set a property (Exchange.SLIP_ENDPOINT) on the Exchange which contains the current endpoint as it advanced though the slip. This allows you to know how far we have processed in the slip. (It's a slip because the Section 2.14, "Dynamic Router" implementation is based on top of Section 2.42, "Routing Slip").

2.14.1.1. Java DSL

In Java DSL you can use the `routingSlip` as shown below:

```
from("direct:start")
    // use a bean as the dynamic router
    .dynamicRouter(bean(DynamicRouterTest.class, "slip"));
```

Which will leverage a Section 3.3, "Bean" to compute the slip *on-the-fly*, which could be implemented as follows:

```
/**
 * Use this method to compute dynamic where we should route next.
 *
 * @param body the message body
 * @return endpoints to go, or
<tt>null</tt>
 to indicate the end
 */
public String slip(String body) {
    bodies.add(body);
    invoked++;

    if (invoked == 1) {
        return "mock:a";
    } else if (invoked == 2) {
        return "mock:b,mock:c";
    } else if (invoked == 3) {
        return "direct:foo";
    } else if (invoked == 4) {
        return "mock:result";
    }

    // no more so return null
    return null;
}
```

Mind that this example is only for show and tell. The current implementation is not thread safe. You would have to store the state on the Exchange, to ensure thread safety.

2.14.1.2. Spring XML

The same example in Spring XML would be:

```xml
<bean id="mySlip" class="org.apache.camel.processor.DynamicRouterTest"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <dynamicRouter>
            <!-- use a method call on a bean as dynamic router -->
            <method ref="mySlip" method="slip"/>
        </dynamicRouter>
    </route>

    <route>
        <from uri="direct:foo"/>
        <transform><constant>Bye World</constant></transform>
        <to uri="mock:foo"/>
    </route>

</camelContext>
```

2.14.1.3. @DynamicRouter annotation

You can also use the `@DynamicRouter` annotation, for example the example below could be written as follows. The `route` method would then be invoked repeatedly as the message is processed dynamically. The idea is to return the next endpoint uri where to go. Return `null` to indicate the end. You can return multiple endpoints if you like, just as the Section 2.42, "Routing Slip", where each endpoint is separated by a delimiter.

```java
public class MyDynamicRouter {

    @Consume(uri = "activemq:foo")
    @DynamicRouter
    public String route(@XPath("/customer/id") String customerId,
    @Header("Location") String location, Document body) {
        // query a database to find the best match of the endpoint based on
        // the input parameters
        // return the next endpoint uri, where to go. Return null to indicate
        // the end.
    }
}
```

2.15. Event Driven Consumer

Camel supports the Event Driven Consumer from the EIP patterns. The default consumer model is event based (i.e. asynchronous) as this means that the Camel container can then manage pooling, threading and concurrency for you in a declarative manner.
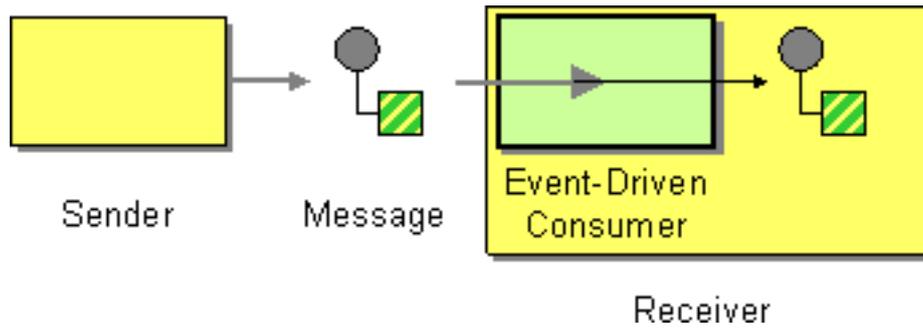
The Event Driven Consumer is implemented by consumers implementing the Processor interface which is invoked by the Section 2.26, "Message Endpoint" when a Section 2.22, "Message" is available for processing.

For more details see

- Section 2.22, "Message"

- Section 2.26, "Message Endpoint"

2.16. Event Message

Camel supports the Event Message from the EIP patterns by supporting the Exchange Pattern on a Section 2.22, "Message" which can be set to **InOnly** to indicate a oneway event message. Camel Components then implement this pattern using the underlying transport or protocols.

> **Note**
>
> See also the related Section 2.39, "Request Reply" EIP.
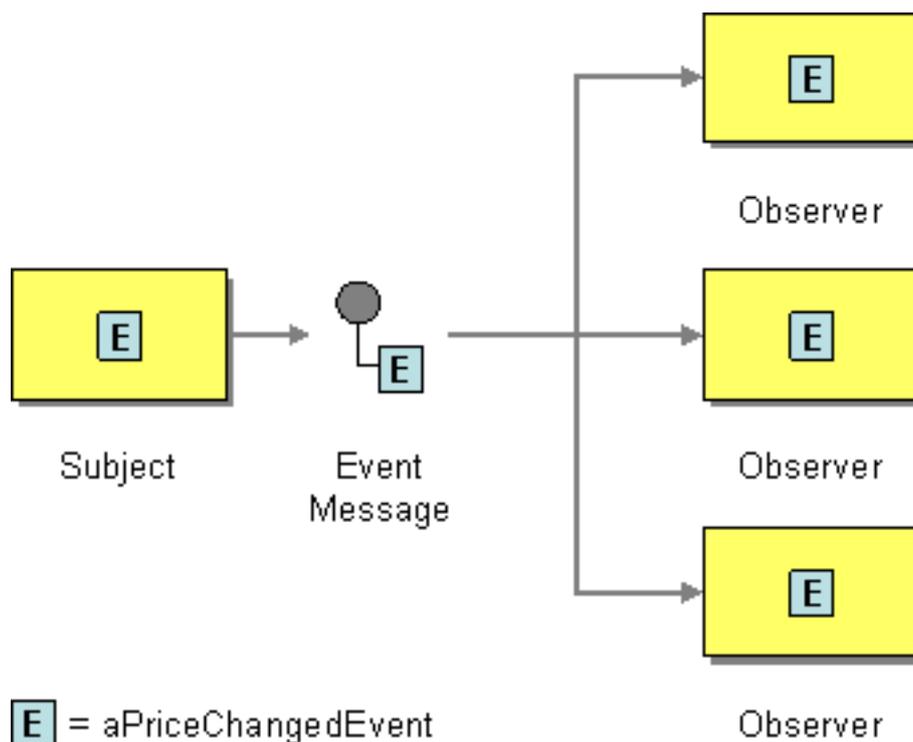
The default behavior of many Components is InOnly such as for Section 3.25, "JMS" or Section 3.39, "SEDA"

2.16.1. Explicitly specifying InOnly

If you are using a component which defaults to InOut you can override the Exchange Pattern for an endpoint using the pattern property.

```
foo:bar?exchangePattern=InOnly
```

From 2.0 onwards on Camel you can specify the Exchange Pattern using the dsl. Using the Fluent Builders:

```
from("mq:someQueue").
  inOnly().
  bean(Foo.class);
```

or you can invoke an endpoint with an explicit pattern

```
<route>
<from uri="mq:someQueue"/>
<inOnly uri="bean:foo"/>
</route>

<route>
<from uri="mq:someQueue"/>
<inOnly uri="mq:anotherQueue"/>
</route>
```

Using the Spring XML Extensions:

```
from("mq:someQueue").
  inOnly().
  bean(Foo.class);
```

2.17. Guaranteed Delivery

Camel supports the Guaranteed Delivery from the EIP patterns using the following components

- ??? for using file systems as a persistent store of messages

- Section 3.25, "JMS" when using persistent delivery (the default) for working with JMS Queues and Topics for high performance, clustering and load balancing

- Section 3.27, "JPA" for using a database as a persistence layer, or use any of the many other database components such as SQL, JDBC, iBatis/MyBatis, Hibernate

- HawtDB for a lightweight key-value persistent store
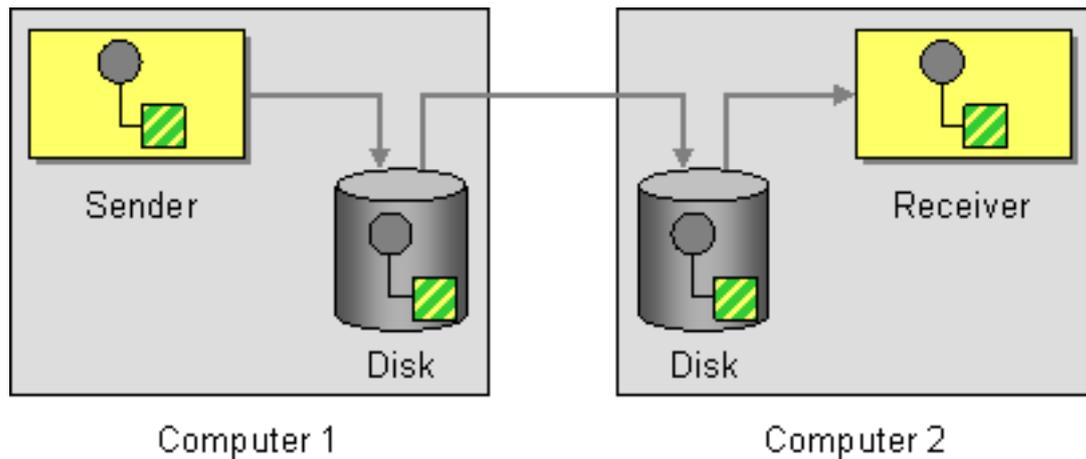
2.18. Idempotent Consumer

2.18.1. Idempotent Consumer

The Idempotent Consumer from the EIP patterns is used to filter out duplicate messages.

This pattern is implemented using the IdempotentConsumer class. This uses an Expression to calculate a unique message ID string for a given message exchange; this ID can then be looked up in the IdempotentRepository to see if it has been seen before; if it has the message is consumed; if it is not then the message is processed and the ID is added to the repository.

The Idempotent Consumer essentially acts like a Section 2.27, "Message Filter" to filter out duplicates.

Camel will add the message id eagerly to the repository to detect duplication also for Exchanges currently in progress. On completion Camel will remove the message id from the repository if the Exchange failed, otherwise it stays there.

Camel provides the following Idempotent Consumer implementations:

- MemoryIdempotentRepository

- FileIdempotentRepository

- JdbcMessageIdRepository (**Available as of Camel 2.7**)

- JpaMessageIdRepository

2.18.2. Options

The Idempotent Consumer has the following options:

Option	Default	Description
eager	true	Eager controls whether Camel adds the message to the repository before or after the exchange has been processed. If enabled before then Camel will be able to detect duplicate messages even when messages are currently in progress. By

Option	Default	Description
		disabling Camel will only detect duplicates when a message has successfully been processed.
messageIdRepositoryRef	`null`	A reference to a `IdempotentRepository` to lookup in the registry. This option is mandatory when using XML DSL.
removeOnFailure	`true`	Sets whether to remove the id of an Exchange that failed.

2.18.3. Using the Fluent Builders

The following example will use the header **myMessageId** to filter out duplicates

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("mock:error"));

        from("seda:a")
            .idempotentConsumer(header("myMessageId"),
                MemoryIdempotentRepository.memoryIdempotentRepository(200))
            .to("seda:b");
    }
};
```

The above example will use an in-memory based MessageIdRepository which can easily run out of memory and doesn't work in a clustered environment. So you might prefer to use the JPA based implementation which uses a database to store the message IDs which have been processed

```
from("direct:start").idempotentConsumer(
        header("messageId"),
        jpaMessageIdRepository(lookup(JpaTemplate.class), PROCESSOR_NAME)
).to("mock:result");
```

In the above example we are using the header **messageId** to filter out duplicates and using the collection **myProcessorName** to indicate the Message ID Repository to use. This name is important as you could process the same message by many different processors; so each may require its own logical Message ID Repository.

For further examples of this pattern in use you could look at the junit test case

2.18.4. Spring XML example

The following example will use the header **myMessageId** to filter out duplicates

```
<!-- repository for the idempotent consumer -->
<bean id="myRepo"
class="org.apache.camel.processor.idempotent.MemoryIdempotentRepository"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <idempotentConsumer messageIdRepositoryRef="myRepo">
            <!-- use the messageId header as key for identifying duplicate
                messages -->
            <header>messageId</header>
            <!-- if not a duplicate send it to this mock endpoint -->
```

```
            <to uri="mock:result"/>
        </idempotentConsumer>
    </route>
</camelContext>
```

2.19. Load Balancer

The Load Balancer Pattern allows you to delegate to one of a number of endpoints using a variety of different load balancing policies.

2.19.1. Built-in load balancing policies

Camel provides the following policies out-of-the-box:

Policy	Description
Round Robin	The exchanges are selected from in a round robin fashion. This is a well known and classic policy, which spreads the load evenly.
Random	A random endpoint is selected for each exchange.
Sticky	Sticky load balancing using an Expression to calculate a correlation key to perform the sticky load balancing; rather like jsessionid in the web or JMSXGroupID in JMS.
Topic	Topic which sends to all destinations (rather like JMS Topics).
Failover	In case of failures the exchange is tried on the next endpoint.
Weighted Round Robin	The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to others.In addition to the weight, endpoint selection is then further refined using **round-robin** distribution based on weight.
Weighted Random	The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to others.In addition to the weight, endpoint selection is then further refined using **random** distribution based on weight.
Custom	The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to others.In addition to the weight, endpoint selection is then further refined using **random** distribution based on weight.

2.19.2. Round Robin

The round robin load balancer is not meant to work with failover, for that you should use the dedicated **failover** load balancer. The round robin load balancer will only change to next endpoint per message.

The round robin load balancer is stateful as it keeps state which endpoint to use next time.

Using the Fluent Builders

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

Using the Spring configuration

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

So the above example will load balance requests from **direct:start** to one of the available **mock endpoint** instances, in this case using a round robbin policy. For further examples of this pattern in use you could look at the junit test case

2.19.3. Failover

The `failover` load balancer is capable of trying the next processor in case an Exchange failed with an `exception` during processing. You can configure the `failover` with a list of specific exception to only failover. If you do not specify any exceptions it will failover over any exceptions. It uses the same strategy for matching exceptions as the Exception Clause does for the **onException**.

⚠️ **Enable stream caching if using streams**

> If you use streaming then you should enable Stream Caching when using the failover load balancer. This is needed so the stream can be re-read when failing over.

It has the following options:

Option	Type	Default	Description
inheritErrorHandler	boolean	true	Whether or not the Error Handler configured on the route should be used or not. You can disable it if you want the failover to trigger immediately and failover to the next endpoint. On the other hand if you have this option enabled, then Camel will first let the Error Handler try to process the message. The Error Handler may have been configured to redelivery and use delays between attempts. If you have enabled a number of redeliveries then Camel will try to redeliver to the **same** endpoint, and only failover to the next endpoint, when the Error Handler is exhausted.
maximumFailover-Attempts	int	-1	A value to indicate after X failver attempts we should exhaust (give up). Use -1 to indicate newer give up and always try to failover. Use 0 to newer failover. And use e.g. 3 to failover at most 3 times before giving up. This option can be used whether or not round robin is enabled or not.
roundRobin	boolean	false	Whether or not the `failover` load balancer should operate in round robin mode or not. If not, then it will **always** start from the first endpoint when a new message is to be processed. In other words it restart from the

Option	Type	Default	Description
			top for every message. If round robin is enabled, then it keeps state and will continue with the next endpoint in a round robin fashion. When using round robin it will not *stick* to last known good endpoint, it will always pick the next endpoint to use.

The `failover` load balancer supports round robin mode, which allows you to failover in a round robin fashion. See the `roundRobin` option.

Here is a sample to failover only if a `IOException` related exception was thrown:

```
from("direct:start")
    // here we will load balance if IOException was thrown
    // any other kind of exception will result in the Exchange as failed
    // to failover over any kind of exception we can just omit the exception
    // in the failOver DSL
    .loadBalance().failover(IOException.class)
        .to("direct:x", "direct:y", "direct:z");
```

You can specify multiple exceptions to failover as the option is varargs, for instance:

```
// enable redelivery so failover can react
errorHandler(defaultErrorHandler().maximumRedeliveries(5));

from("direct:foo").
    loadBalance().failover(IOException.class, MyOtherException.class)
        .to("direct:a", "direct:b");
```

2.19.3.1. Using failover in Spring DSL

Failover can also be used from Spring DSL and you configure it as:

```
<route errorHandlerRef="myErrorHandler">
      <from uri="direct:foo"/>
      <loadBalance>
          <failover>
              <exception>java.io.IOException</exception>
              <exception>com.mycompany.MyOtherException</exception>
          </failover>
          <to uri="direct:a"/>
          <to uri="direct:b"/>
      </loadBalance>
    </route>
```

2.19.3.2. Using failover in round robin mode

An example using Java DSL:

```
from("direct:start")
    // Use failover load balancer in stateful round robin mode
    // which mean it will failover immediately in case of an exception
    // as it does NOT inherit error handler. It will also keep retrying as
    // it is configured to newer exhaust.
```

```
        .loadBalance().failover(-1, false, true).
            to("direct:bad", "direct:bad2", "direct:good", "direct:good2");
```

And the same example using Spring XML:

```
<route>
    <from uri="direct:start"/>
    <loadBalance>
        <!-- failover using stateful round robin,
             which will keep retrying forever those
             4 endpoints until success. You can set
             the maximumFailoverAttempt to break out after
             X attempts -->
        <failover roundRobin="true"/>
        <to uri="direct:bad"/>
        <to uri="direct:bad2"/>
        <to uri="direct:good"/>
        <to uri="direct:good2"/>
    </loadBalance>
</route>
```

2.19.4. Weighted Round-Robin and Random Load Balancing

In many enterprise environments where server nodes of unequal processing power & performance characteristics are utilized to host services and processing endpoints, it is frequently necessary to distribute processing load based on their individual server capabilities so that some endpoints are not unfairly burdened with requests. Obviously simple round-robin or random load balancing do not alleviate problems of this nature. A Weighted Round-Robin and/or Weighted Random load balancer can be used to address this problem.

The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to others. You can specify this as a positive processing weight for each server. A larger number indicates that the server can handle a larger load. The weight is utilized to determine the payload distribution ratio to different processing endpoints with respect to others.

The parameters that can be used are

Option	Type	Default	Description
roundRobin	boolean	false	The default value for round-robin is false. In the absence of this setting or parameter the load balancing algorithm used is random.
distributionRatio	String	none	The distributionRatio is a delimited String consisting on integer weights separated by delimiters for example "2,3,5". The distributionRatio must match the number of endpoints and/or processors specified in the load balancer list.
distributionRatio-Delimiter	String	,	The distributionRatioDelimiter is the delimiter used to specify the distributionRatio. If this attribute is not specified a default delimiter "," is expected as the delimiter used for specifying the distributionRatio.

See the Camel website for examples on using this load balancer.

2.20. Log

How can I log processing a Section 2.22, "Message" ?

Camel provides many ways to log processing a message. Here is just some examples:

• You can use the Section 3.29, "Log" component which logs the Message content.

• You can use the Tracer which trace logs message flow.

• You can also use a Processor or Section 3.3, "Bean" and log from Java code.

• You can use the `log` DSL.

2.20.1. Using log DSL

You can use the `log` DSL which allows you to use Simple language to construct a dynamic message which gets logged. For example you can do

```
from("direct:start").log("Processing ${id}").
   to("bean:foo");
```

Which will construct a String message at runtime using the Simple language. The log message will by logged at `INFO` level using the route id as the log name. By default a route is named `route-1`, `route-2` etc. But you can use the `routeId("myCoolRoute")` to set a route name of choice.

> ⚠ **Difference between log in the DSL and Log component**
>
> The `log` DSL is much lighter and meant for logging human logs such as `Starting   to do  ...` and so on. It can only log a message based on the Simple language. On the other hand Section 3.29, "Log" component is a full fledged component which involves using endpoints and etc. The Section 3.29, "Log" component is meant for logging the Message itself and you have many URI options to control what you would like to be logged.

The log DSL have overloaded methods to set the logging level and/or name as well.

```
from("direct:start").log(LoggingLevel.DEBUG, "Processing ${id}").to("bean:foo");
```

For example you can use this to log the file name being processed if you consume files.

```
from("file://target/files").log(LoggingLevel.DEBUG,
   "Processing file ${file:name}").to("bean:foo");
```

2.20.1.1. Using log DSL from Spring

In Spring DSL it is also easy to use log DSL as shown below:

```
<route id="foo">
        <from uri="direct:foo"/>
        <log message="Got ${body}"/>
        <to uri="mock:foo"/>
    </route>
```

The log tag has attributes to set the `message`, `loggingLevel` and `logName`. For example:

```
<route id="baz">
     <from uri="direct:baz"/>
     <log message="Me Got ${body}" loggingLevel="FATAL" logName="cool"/>
     <to uri="mock:baz"/>
</route>
```

2.20.1.2. Using slf4j Marker

You can specify a marker name in the DSL:

```
<route id="baz">
    <from uri="direct:baz"/>
    <log message="Received ${body}" loggingLevel="FATAL" logName="cool"
        marker="myMarker"/>
    <to uri="mock:baz"/>
</route>
```

2.21. Loop

The Loop allows for processing a message a number of times, possibly in a different way for each iteration. Useful mostly during testing. Options:

Name	Default Value	Description
copy	false	Whether or not copy mode is used. If false then the same Exchange will be used for each iteration. So the result from the previous iteration will be visible for the next iteration. Instead you can enable copy mode, and then each iteration restarts with a fresh copy of the input Exchange.

For each iteration two properties are set on the Exchange. These properties can be used by processors down the pipeline to process the Section 2.22, "Message" in different ways.

Property	Description
CamelLoopSize	Total number of loops
CamelLoopIndex	Index of the current iteration (0 based)

that could be used by processors down the pipeline to process the Section 2.22, "Message" in different ways.

2.21.1. Examples

The following example shows how to take a request from the **direct:x** endpoint, then send the message repetitively to **mock:result**. The number of times the message is sent is either passed as an argument to loop(), or determined at runtime by evaluating an expression. The expression **must** evaluate to an int, otherwise a RuntimeCamelException is thrown.

Using the Fluent Builders

Pass loop count as an argument

```
from("direct:a").loop(8).to("mock:result");
```

Use expression to determine loop count

```
from("direct:b").loop(header("loop")).to("mock:result");
```

Use expression to determine loop count

```
from("direct:c").loop().xpath("/hello/@times").to("mock:result");
```

Using the Spring XML Extensions

Pass loop count as an argument

```
<route>
  <from uri="direct:a"/>
  <loop>
    <constant>8</constant>
    <to uri="mock:result"/>
  </loop>
</route>
```

Use expression to determine loop count

```
<route>
  <from uri="direct:b"/>
  <loop>
    <header>loop</header>
    <to uri="mock:result"/>
  </loop>
</route>
```

See the Camel Website for further examples of this pattern in use.

2.22. Message

Camel supports the Message from the EIP patterns using the Message interface.
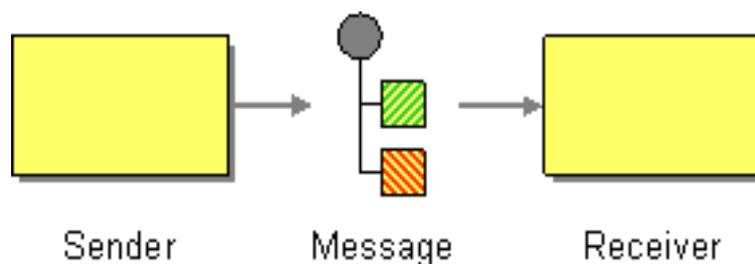
To support various message exchange patterns like one way Section 2.16, "Event Message" and Section 2.39, "Request Reply" messages Camel uses an Exchange interface which has a **pattern** property which can be set to **InOnly** for an Section 2.16, "Event Message" which has a single inbound Message, or **InOut** for a Section 2.39, "Request Reply" where there is an inbound and outbound message.

Here is a basic example of sending a Message to a route in InOnly and InOut modes

Requestor Code

```
//InOnly
getContext().createProducerTemplate().sendBody("direct:startInOnly",
    "Hello World");

//InOut
```

```
String result = (String) getContext().createProducerTemplate().requestBody(
    "direct:startInOut", "Hello World");
```

Route Using the Fluent Builders

```
from("direct:startInOnly").inOnly("bean:process");

from("direct:startInOut").inOut("bean:process");
```

Route Using the Spring XML Extensions

```
<route>
  <from uri="direct:startInOnly"/>
  <inOnly uri="bean:process"/>
</route>

<route>
  <from uri="direct:startInOut"/>
  <inOut uri="bean:process"/>
</route>
```

2.23. Message Bus

Camel supports the Message Bus from the EIP patterns. You could view Camel as a Message Bus itself as it allows producers and consumers to be decoupled.
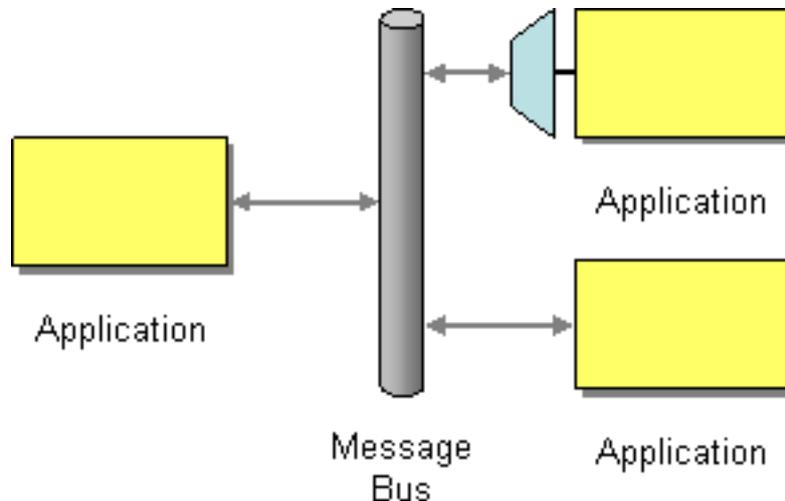
Folks often assume that a Message Bus is a JMS though so you may wish to refer to the Section 3.25, "JMS" component for traditional MOM support.

Also worthy of node is the ??? component for supporting messaging over XMPP (Jabber)

2.24. Message Channel

Camel supports the Message Channel from the EIP patterns. The Message Channel is an internal implementation detail of the Endpoint interface and all interactions with the Message Channel are via the Endpoint interfaces. For more details see Section 2.22, "Message" and Section 2.26, "Message Endpoint".
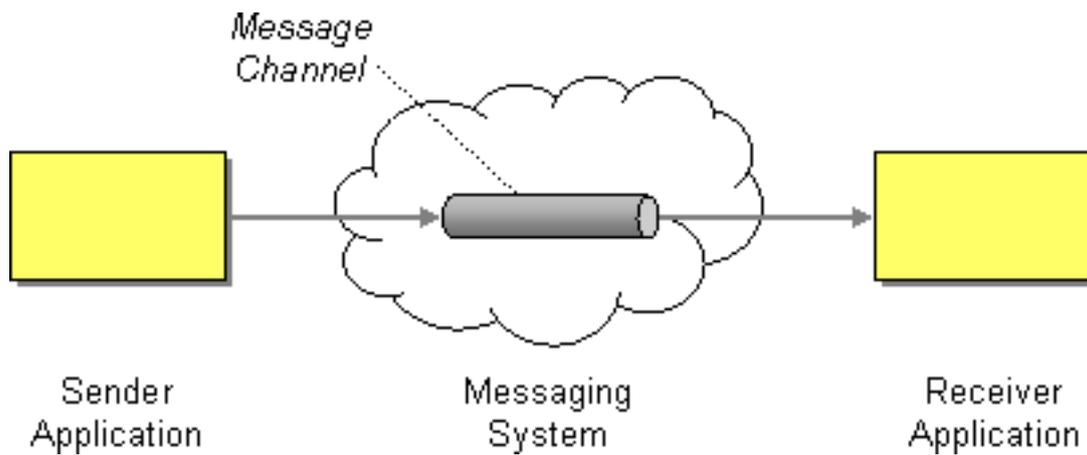
2.25. Message Dispatcher

Camel supports the Message Dispatcher from the EIP patterns using various approaches.
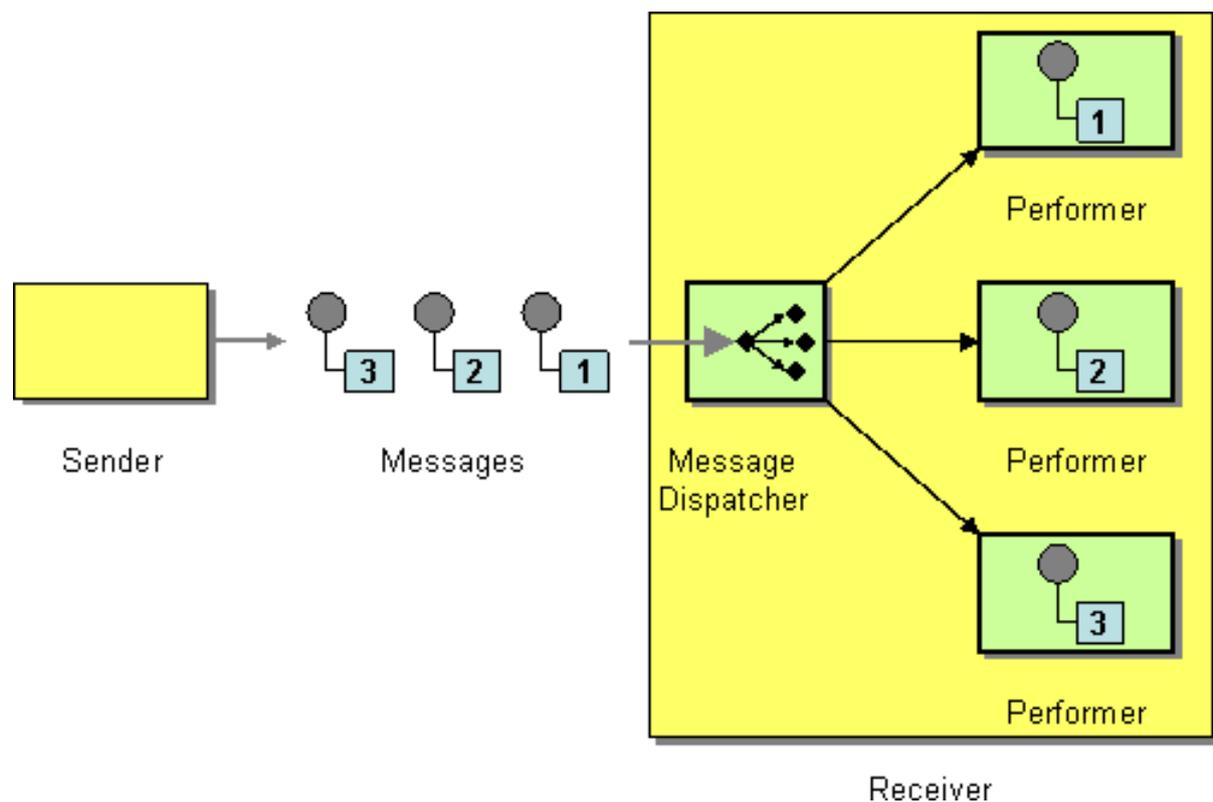
You can use a component like Section 3.25, "JMS" with selectors to implement a Section 2.45, "Selective Consumer" as the Message Dispatcher implementation. Or you can use an Endpoint as the Message Dispatcher itself and then use a Section 2.6, "Content Based Router" as the Message Dispatcher.

2.26. Message Endpoint

Camel supports the Message Endpoint from the EIP patterns using the Endpoint interface.
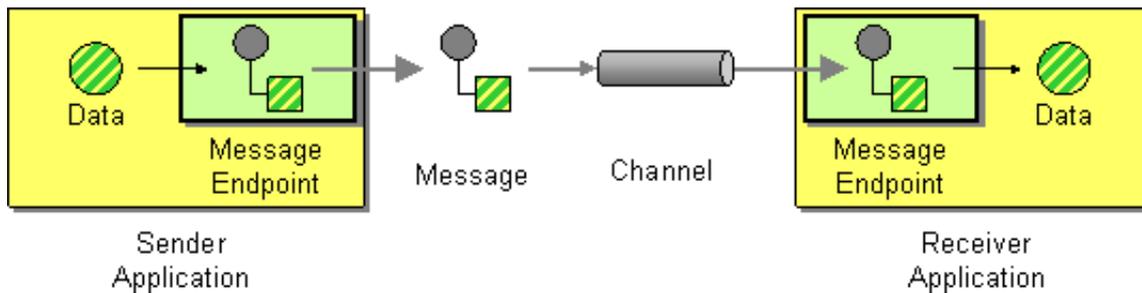
When using the DSL to create Routes you typically refer to Message Endpoints by their URIs rather than directly using the Endpoint interface. it is then a responsibility of the CamelContext to create and activate the necessary Endpoint instances using the available Component implementations.

2.27. Message Filter

The Message Filter from the EIP patterns allows you to filter messages
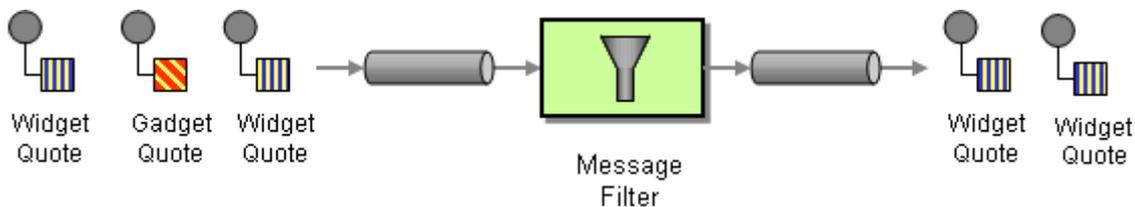
The following example shows how to create a Message Filter route consuming messages from an endpoint called **queue:a** which if the Predicate is true will be dispatched to **queue:b**

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("mock:error"));

        from("seda:a")
            .filter(header("foo").isEqualTo("bar"))
                .to("seda:b");
    }
};
```

You can of course use many different Predicate languages such as XPath, XQuery, SQL or various Scripting Languages. Here is an XPath example

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

Using the Spring XML Extensions

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

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

For further examples of this pattern in use you could look at the junit test case

2.27.1. Using stop

Stop is a bit different than a message filter as it will filter out all messages. Stop is convenient to use in a Section 2.6, "Content Based Router" when you for example need to stop further processing in one of the predicates.

In the example below we do not want to route messages any further that has the word `Bye` in the message body. Notice how we prevent this in the when predicate by using the `.stop()`.

```
from("direct:start")
    .choice()
        .when(body().contains("Hello")).to("mock:hello")
        .when(body().contains("Bye")).to("mock:bye").stop()
        .otherwise().to("mock:other")
    .end()
    .to("mock:result");
```

2.27.2. Knowing if Exchange was filtered or not

The Section 2.27, "Message Filter" EIP will add a property on the Exchange which states if it was filtered or not.

The property has the key `Exchannge.FILTER_MATCHED` which has the String value of `CamelFilterMatched`. Its value is a boolean indicating `true` or `false`. If the value is `true` then the Exchange was routed in the filter block.

2.28. Message Router

The Message Router from the EIP patterns allows you to consume from an input destination, evaluate some predicate then choose the right output destination.
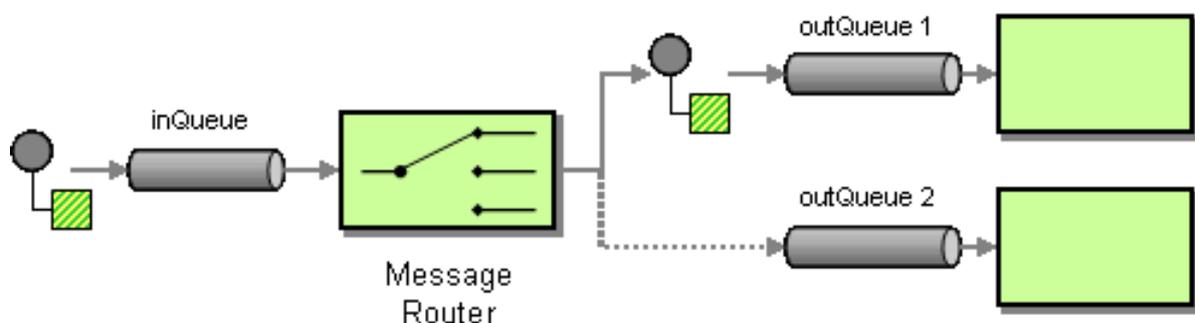
The following example shows how to route a request from an input **queue:a** endpoint to either **queue:b**, **queue:c** or **queue:d** depending on the evaluation of various Predicate expressions

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("mock:error"));

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

Here is another example of using a bean to define the filter behavior

```
from("direct:start")
.filter().method(MyBean.class, "isGoldCustomer").to("mock:result").end()
.to("mock:end");

public static class MyBean {
    public boolean isGoldCustomer(@Header("level") String level) {
        return level.equals("gold");
    }
}
```

Using the Spring XML Extensions

```
<camelContext errorHandlerRef="errorHandler"
    xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <choice>
            <when>
                <xpath>$foo = 'bar'</xpath>
                <to uri="seda:b"/>
            </when>
            <when>
                <xpath>$foo = 'cheese'</xpath>
                <to uri="seda:c"/>
            </when>
            <otherwise>
                <to uri="seda:d"/>
            </otherwise>
        </choice>
    </route>
</camelContext>
```

Note if you use a `choice` without adding an `otherwise`, any unmatched exchanges will be dropped by default.

2.29. Message Translator

Camel supports the Message Translator from the EIP patterns by using an arbitrary Processor in the routing logic, by using a bean to perform the transformation, or by using transform() in the DSL. You can also use a Data Format to marshal and unmarshal messages in different encodings.

Translator

Incoming Message — Translated Message
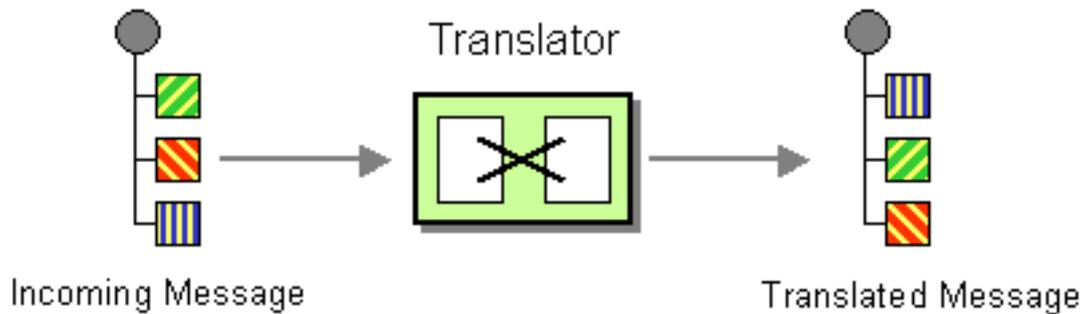
Using the Fluent Builders

You can transform a message using Camel's Bean Integration to call any method on a bean in your Registry such as your Spring XML configuration file as follows

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

Where the "myTransformerBean" would be defined in a Spring XML file or defined in JNDI and so on. You can omit the method name parameter from beanRef() and the Bean Integration will try to deduce the method to invoke from the message exchange.

or you can add your own explicit Processor to do the transformation

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

or you can use the DSL to explicitly configure the transformation

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

Use Spring XML

You can also use Spring XML Extensions to do a transformation. Basically any Expression language can be substituted inside the transform element as shown below

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <transform>
      <simple>${in.body} extra data!</simple>
    </transform>
    <to uri="mock:end"/>
  </route>
</camelContext>
```

Or you can use the Bean Integration to invoke a bean

```
<route>
  <from uri="activemq:Input"/>
  <bean ref="myBeanName" method="doTransform"/>
  <to uri="activemq:Output"/>
</route>
```

You can also use Templating to consume a message from one destination, transform it with something like Section 3.51, "Velocity" or XQuery and then send it on to another destination. For example using InOnly (one way messaging)

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

If you want to use InOut (request-reply) semantics to process requests on the **My.Queue** queue on ??? with a template generated response, then sending responses back to the JMSReplyTo Destination you could use this.

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

2.30. Messaging Gateway

Camel has several endpoint components that support the Messaging Gateway from the EIP patterns.
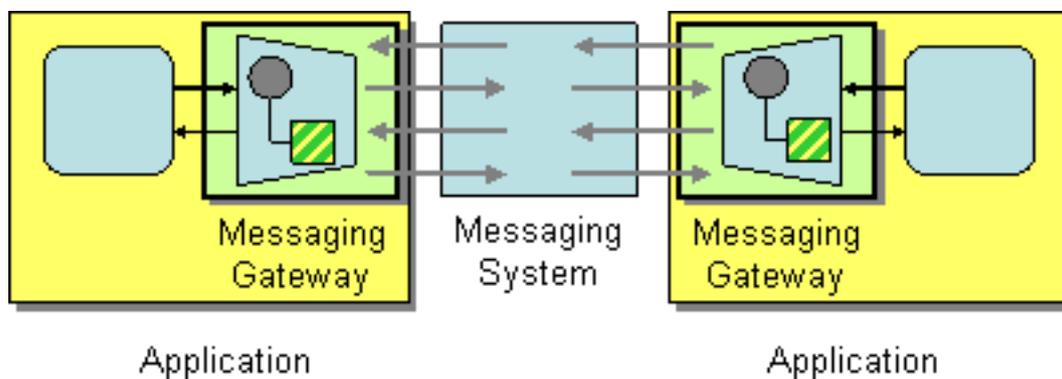
Components like Section 3.3, "Bean" and Section 3.8, "CXF" provide a a way to bind a Java interface to the message exchange.

However you may want to read the Using CamelProxy documentation as a true Section 2.30, "Messaging Gateway" EIP solution. Another approach is to use @Produce which you can read about in POJO Producing which also can be used as a Section 2.30, "Messaging Gateway" EIP solution.

2.31. Messaging Mapper

Camel supports the Messaging Mapper from the EIP patterns by using either Section 2.29, "Message Translator" pattern or the Type Converter module.
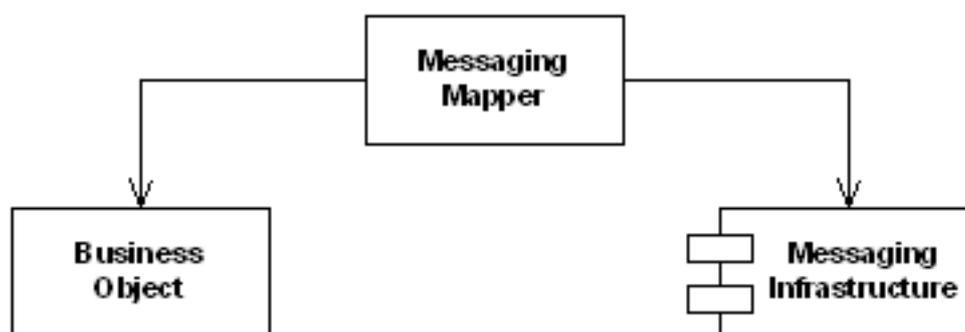
2.32. Multicast

2.32.1. Multicast

The Multicast allows for routing the same message to a number of endpoints and process them in a different way. The main difference between the Multicast and Splitter is that Splitter will split the message into several pieces but the Multicast will not modify the request message.

2.32.1.1. Options

Name	Default Value	Description
strategyRef		Refers to an AggregationStrategy to be used to assemble the replies from the multicasts, into a single outgoing message from the Multicast. By default Camel will use the last reply as the outgoing message.
parallelProcessing	false	If enabled then sending messages to the multicasts occurs concurrently. Note the caller thread will still wait until all messages has been fully processed, before it continues. Its only the sending and processing the replies from the multicasts which happens concurrently.
executorServiceRef		Refers to a custom Thread Pool to be used for parallel processing. Notice if you set this option, then parallel processing is automatic implied, and you do not have to enable that option as well.
stopOnException	false	Whether or not to stop continue processing immediately when an exception occurred. If disabled, then Camel will send the message to all multicasts regardless if one of them failed. You can deal with exceptions in the AggregationStrategy class where you have full control how to handle that.
streaming	false	If enabled then Camel will process replies out-of-order, eg in the order they come back. If disabled, Camel will process replies in the same order as multicasted.
timeout		Sets a total timeout specified in millis. If the Multicast hasn't been able to send and process all replies within the given timeframe, then the timeout triggers and the Multicast breaks out and continues. Notice if you provide a TimeoutAwareAggregationStrategy then the timeout method is invoked before breaking out.
onPrepareRef		Refers to a custom Processor to prepare the copy of the Exchange each multicast will receive. This allows you to do any custom logic, such as deep-cloning the message payload if that's needed etc.
shareUnitOfWork	false	Whether the unit of work should be shared. See the same option on Splitter for more details.

2.32.1.2. Example

The following example shows how to take a request from the **direct:a** endpoint, then multicast these request to **direct:x**, **direct:y**, **direct:z**.

Using the Fluent Builders

```
from("direct:a").multicast().to("direct:x", "direct:y",
"direct:z");
```

By default Multicast invokes each endpoint sequentially. If parallel processing is desired, simply use

```
from("direct:a").multicast().parallelProcessing().to("direct:x", "direct:y",
    "direct:z");
```

In case of using InOut MEP, an AggregationStrategy is used for aggregating all reply messages. The default is to only use the latest reply message and discard any earlier replies. The aggregation strategy is configurable:

```
from("direct:start")
  .multicast(new MyAggregationStrategy())
  .parallelProcessing().timeout(500).to("direct:a", "direct:b", "direct:c")
  .end()
  .to("mock:result");
```

2.32.2. Stop processing in case of exception

The Section 2.32, "Multicast" will by default continue to process the entire Exchange even in case one of the multicasted messages will thrown an exception during routing. For example if you want to multicast to 3 destinations and the second destination fails by an exception. What Camel does by default is to process the remainder destinations. You have the chance to remedy or handle this in the AggregationStrategy.

But sometimes you just want Camel to stop and let the exception be propagated back, and let the Camel error handler handle it. You can do this by specifying that it should stop in case of an exception occurred. This is done by the stopOnException option as shown below:

```
from("direct:start")
        .multicast()
            .stopOnException().to("direct:foo", "direct:bar", "direct:baz")
        .end()
        .to("mock:result");

        from("direct:foo").to("mock:foo");

        from("direct:bar").process(new MyProcessor()).to("mock:bar");

        from("direct:baz").to("mock:baz");
```

And using XML DSL you specify it as follows:

```
<route>
    <from uri="direct:start"/>
    <multicast stopOnException="true">
        <to uri="direct:foo"/>
        <to uri="direct:bar"/>
        <to uri="direct:baz"/>
    </multicast>
    <to uri="mock:result"/>
</route>
```

```
<route>
    <from uri="direct:foo"/>
    <to uri="mock:foo"/>
</route>

<route>
    <from uri="direct:bar"/>
    <process ref="myProcessor"/>
    <to uri="mock:bar"/>
</route>

<route>
    <from uri="direct:baz"/>
    <to uri="mock:baz"/>
</route>
```

2.32.3. Using onPrepare to execute custom logic when preparing messages

The Multicast will copy the source Exchange and multicast each copy. However the copy is a shallow copy, so in case you have mutateable message bodies, then any changes will be visible by the other copied messages. If you want to use a deep clone copy then you need to use a custom onPrepare which allows you to do this using the Processor interface.

Note that onPrepare can be used for any kind of custom logic which you would like to execute before the Exchange is being multicasted.

The Multicast EIP page on the Camel website hosts a dynamically updated example of using onPrepare to execute custom logic.

2.33. Normalizer

Camel supports the Normalizer from the EIP patterns by using a Section 2.28, "Message Router" in front of a number of Section 2.29, "Message Translator" instances.
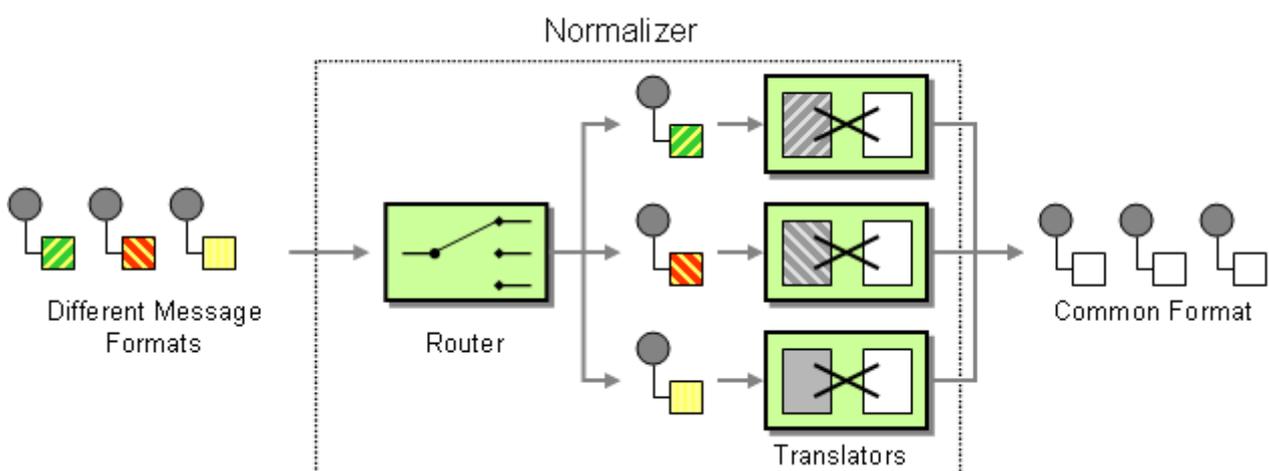
2.33.1. Example

This example shows a Message Normalizer that converts two types of XML messages into a common format. Messages in this common format are then filtered.

Using the Fluent Builders

```
// we need to normalize two types of incoming messages
from("direct:start")
    .choice()
        .when().xpath("/employee").to(
            "bean:normalizer?method=employeeToPerson")
        .when().xpath("/customer").to(
            "bean:normalizer?method=customerToPerson")
    .end()
    .to("mock:result");
```

In this case we're using a Java bean as the normalizer. The class looks like this

```
public class MyNormalizer {
    public void employeeToPerson(Exchange exchange,
        @XPath("/employee/name/text()") String name) {
        exchange.getOut().setBody(createPerson(name));
    }

    public void customerToPerson(Exchange exchange,
        @XPath("/customer/@name") String name) {
        exchange.getOut().setBody(createPerson(name));
    }

    private String createPerson(String name) {
        return "<person name=\"" + name + "\"/>";
    }
}
```

Using the Spring XML Extensions

The same example in the Spring DSL

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <choice>
      <when>
        <xpath>/employee</xpath>
        <to uri="bean:normalizer?method=employeeToPerson"/>
      </when>
      <when>
        <xpath>/customer</xpath>
        <to uri="bean:normalizer?method=customerToPerson"/>
      </when>
    </choice>
    <to uri="mock:result"/>
  </route>
</camelContext>

<bean id="normalizer" class="org.apache.camel.processor.MyNormalizer"/>
```

2.34. Pipes and Filters

Camel supports Pipes and Filters from the EIP patterns in various ways.
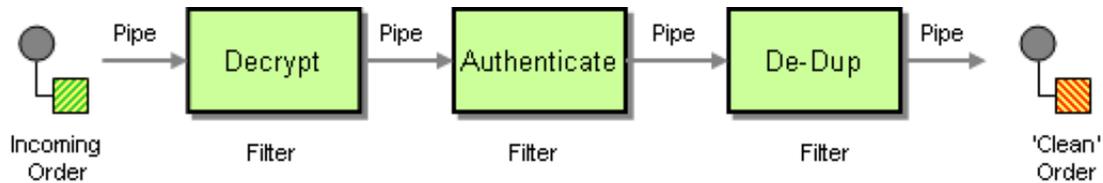
With Camel you can split your processing across multiple independent Endpoint instances which can then be chained together.

2.34.1. Using Routing Logic

You can create pipelines of logic using multiple Endpoint or Section 2.29, "Message Translator" instances as follows

```
from("direct:a").pipeline("direct:x", "direct:y", "direct:z",
    "mock:result");
```

Though pipeline is the default mode of operation when you specify multiple outputs in Camel. The opposite to pipeline is multicast; which fires the same message into each of its outputs. (See the example below).

In Spring XML you can use the <pipeline/> element as of 1.4.0 onwards

```
<route>
  <from uri="activemq:SomeQueue"/>
  <pipeline>
    <bean ref="foo"/>
    <bean ref="bar"/>
    <to uri="activemq:OutputQueue"/>
  </pipeline>
</route>
```

In the above the pipeline element is actually unnecessary, you could use this:

```
<route>
  <from uri="activemq:SomeQueue"/>
  <bean ref="foo"/>
  <bean ref="bar"/>
  <to uri="activemq:OutputQueue"/>
</route>
```

Which is a bit more explicit. However if you wish to use <multicast/> to avoid a pipeline - to send the same message into multiple pipelines - then the <pipeline/> element comes into its own.

```
<route>
  <from uri="activemq:SomeQueue"/>
  <multicast>
    <pipeline>
      <bean ref="something"/>
      <to uri="log:Something"/>
```

```
    </pipeline>
    <pipeline>
      <bean ref="foo"/>
      <bean ref="bar"/>
      <to uri="activemq:OutputQueue"/>
    </pipeline>
  </multicast>
</route>
```

In the above example we are routing from a single Endpoint to a list of different endpoints specified using URIs.

2.35. Point to Point Channel

Camel supports the Point to Point Channel from the EIP patterns using the following components

- Section 3.39, "SEDA" for in-VM seda based messaging

- Section 3.25, "JMS" for working with JMS Queues for high performance, clustering and load balancing

- Section 3.27, "JPA" for using a database as a simple message queue

- ??? for point-to-point communication over XMPP (Jabber)

- and others
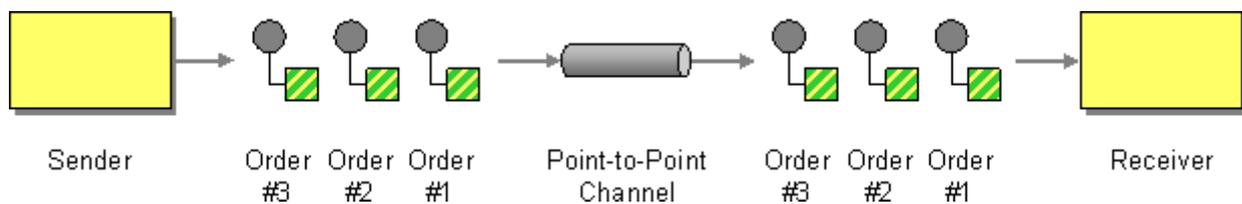
2.36. Polling Consumer

2.36.1. Polling Consumer

Camel supports implementing the Polling Consumer from the EIP patterns using the PollingConsumer interface which can be created via the Endpoint.createPollingConsumer() method.
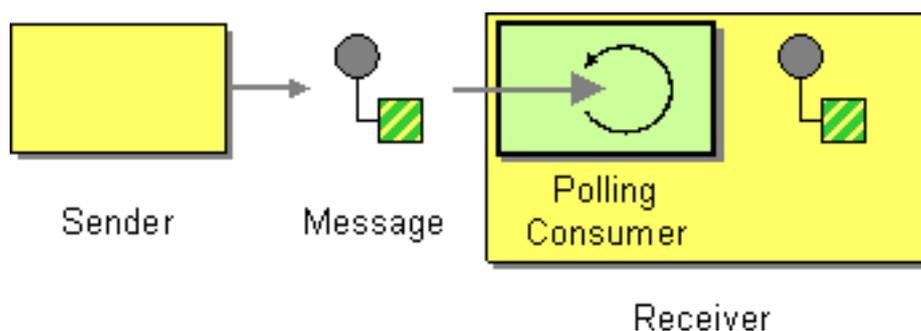
So in your Java code you can do

```
Endpoint endpoint = context.getEndpoint("activemq:my.queue");
PollingConsumer consumer = endpoint.createPollingConsumer();
Exchange exchange = consumer.receive();
```

Notice in Camel 2.0 we have introduced the `ConsumerTemplate`.

There are 3 main polling methods on PollingConsumer

Method name	Description
receive()	Waits until a message is available and then returns it; potentially blocking forever
receive(long)	Attempts to receive a message exchange, waiting up to the given timeout and returning null if no message exchange could be received within the time available
receiveNoWait()	Attempts to receive a message exchange immediately without waiting and returning null if a message exchange is not available yet

2.36.2. ConsumerTemplate

The `ConsumerTemplate` is a template much like Spring's JmsTemplate or JdbcTemplate supporting the Section 2.36, "Polling Consumer" EIP. With the template you can consume Exchange s from an Endpoint.

The template supports the three operations above, but also including convenient methods for returning the body: `consumeBody`, and so on. The example from above using ConsumerTemplate is:

```
Exchange exchange = consumerTemplate.receive("activemq:my.queue");
```

Or to extract and get the body you can do:

```
Object body = consumerTemplate.receiveBody("activemq:my.queue");
```

And you can provide the body type as a parameter and have it returned as the type:

```
String body = consumerTemplate.receiveBody("activemq:my.queue",
    String.class);
```

You get hold of a `ConsumerTemplate` from the `CamelContext` with the `createConsumerTemplate` operation:

```
ConsumerTemplate consumer = context.createConsumerTemplate();
```

For using Spring DSL with **consumerTemplate**, see the dynamically maintained examples for the most up-to-date examples.

2.36.3. Scheduled Poll Components

Quite a few inbound Camel endpoints use a scheduled poll pattern to receive messages and push them through the Camel processing routes. That is to say externally from the client the endpoint appears to use an Section 2.15, "Event Driven Consumer" but internally a scheduled poll is used to monitor some kind of state or resource and then fire message exchanges. Since this is such a common pattern, polling components can extend the ScheduledPollConsumer base class which makes it simpler to implement this pattern.

The ScheduledPollConsumer supports the following options:

Option	Default	Description
pollStrategy		A pluggable org.apache.camel. PollingConsumerPollStrategy allowing you to provide your custom implementation to control error handling usually occurred during the `poll` operation **before** an Exchange have been created and being routed in Camel. In other words the error occurred while the polling was gathering information, for instance access to a file network failed so Camel cannot access it to scan for files. The default implementation will log the caused exception at WARN level and ignore it.
sendEmptyMessageWhenIdle	false	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.

2.36.4. About error handling and scheduled polling consumers

ScheduledPollConsumer is scheduled based and its `run` method is invoked periodically based on schedule settings. But errors can also occur when a poll is being executed. For instance if Camel should poll a file network, and this network resource is not available then a `java.io.IOException` could occur. As this error happens **before** any Exchange has been created and prepared for routing, then the regular Error Handling in Camel does not apply. So what does the consumer do then? Well the exception is propagated back to the `run` method where it is handled. Camel will by default log the exception at WARN level and then ignore it. At next schedule the error could have been resolved and thus being able to poll the endpoint successfully.

2.36.4.1. Controlling the error handling using PollingConsumerPollStrategy

org.apache.camel.PollingConsumerPollStrategy is a pluggable strategy that you can configure on the ScheduledPollConsumer. The default implementation org.apache.camel.impl.DefaultPollingConsumerPollStrategy will log the caused exception at WARN level and then ignore this issue.

The strategy interface provides the following 3 methods

- begin

  - `void begin(Consumer consumer, Endpoint endpoint)`

- begin (**Camel 2.3**)

  - `boolean begin(Consumer consumer, Endpoint endpoint)`

- commit

  - `void commit(Consumer consumer, Endpoint endpoint)`

- commit (**Camel 2.6**)

  - `void commit(Consumer consumer, Endpoint endpoint, int polledMessages)`

- rollback

- boolean rollback(Consumer consumer, Endpoint endpoint, int retryCounter,
  Exception e) throws Exception

The begin method returns a boolean which indicates whether or not to skipping polling. So you can implement your custom logic and return `false` if you do not want to poll this time.

In **Camel 2.6** onwards the commit method has an additional parameter containing the number of message that was actually polled. For example if there was no messages polled, the value would be zero, and you can react accordingly.

The most interesting is the `rollback` as it allows you do handle the caused exception and decide what to do.

For instance if we want to provide a retry feature to a scheduled consumer we can implement the `PollingConsumerPollStrategy` method and put the retry logic in the `rollback` method. Let's just retry up until 3 times:

```
public boolean rollback(Consumer consumer, Endpoint endpoint, int retryCounter,
    Exception e) throws Exception {
    if (retryCounter < 3) {
        // return true to tell Camel that it should retry the poll immediately
        return true;
    }
    // okay we give up do not retry anymore
    return false;
}
```

Notice that we are given the `Consumer` as a parameter. We could use this to *restart* the consumer as we can invoke stop and start:

```
// error occurred let's restart the consumer,
// that could maybe resolve the issue
consumer.stop();
consumer.start();
```

Notice: If you implement the `begin` operation make sure to avoid throwing exceptions as in such a case the `poll` operation is not invoked and Camel will invoke the `rollback` directly.

2.36.4.2. Configuring an Endpoint to use PollingConsumerPollStrategy

To configure an Endpoint to use a custom `PollingConsumerPollStrategy` you use the option `pollStrategy`. For example in the file consumer below we want to use our custom strategy defined in the Registry with the bean id `myPoll`:

```
from("file://inbox/?pollStrategy=#myPoll").to("activemq:queue:inbox")
```

2.37. Publish Subscribe Channel

Camel supports the Publish Subscribe Channel from the EIP patterns using the following components

- Section 3.25, "JMS" for working with JMS Topics for high performance, clustering and load balancing

- ??? when using rooms for group communication
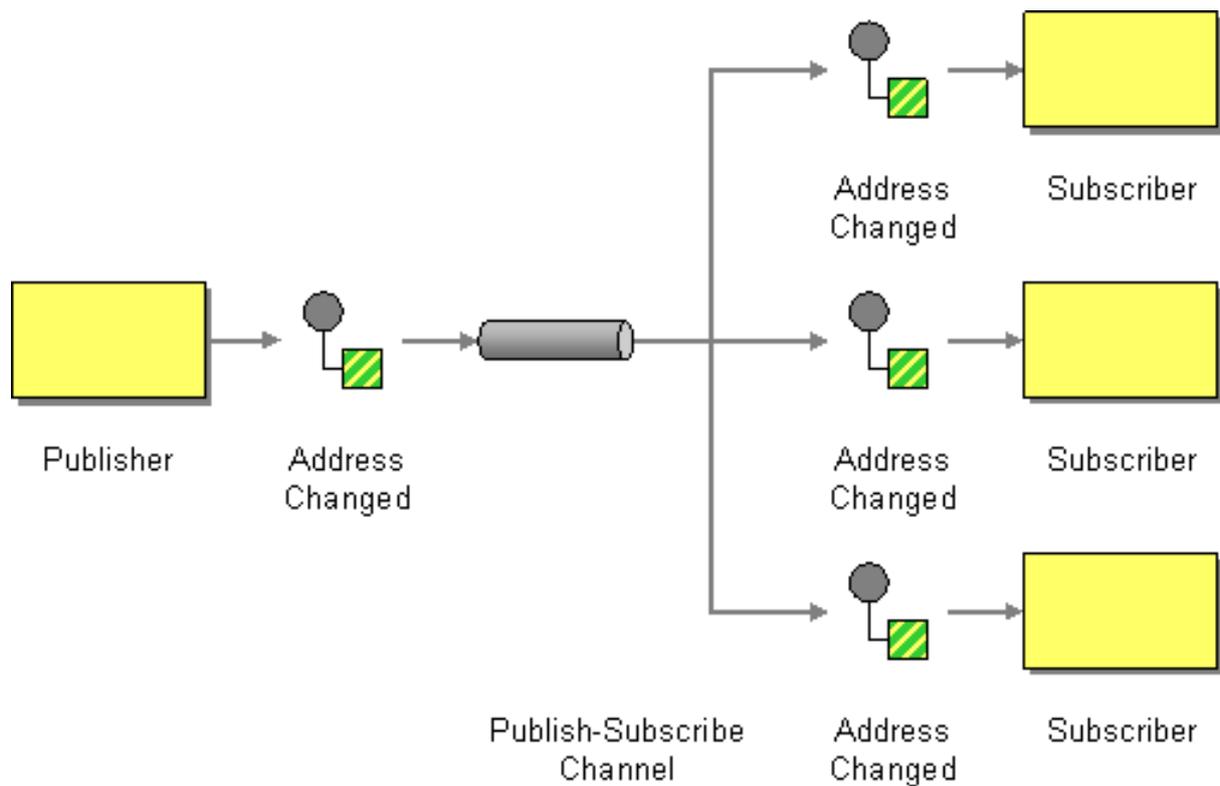
2.37.1. Using Routing Logic

Another option is to explicitly list the publish-subscribe relationship in your routing logic; this keeps the producer and consumer decoupled but lets you control the fine grained routing configuration using the DSL or XML Configuration.

Using the Spring XML Extensions

```
<camelContext errorHandlerRef="errorHandler"
    xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <multicast>
            <to uri="seda:b"/>
            <to uri="seda:c"/>
            <to uri="seda:d"/>
        </multicast>
    </route>
</camelContext>
```

2.38. Recipient List

The Recipient List from the EIP patterns allows you to route messages to a number of dynamically specified recipients.
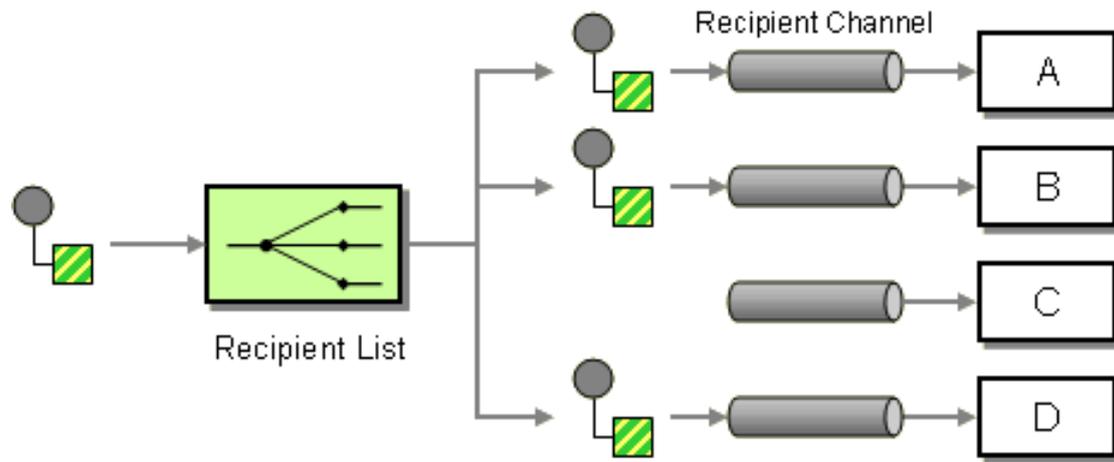
The recipients will receive a copy of the same Exchange and Camel will execute them sequentially.

2.38.1. Options

Name	Default Value	Description
delimiter	,	Delimiter used if the Expression returned multiple endpoints.
strategyRef		An AggregationStrategy that will assemble the replies from recipients into a single outgoing message from the Recipient List. By default Camel will use the last reply as the outgoing message.
parallelProcessing	false	If enabled, messages are sent to the recipients concurrently. Note that the calling thread will still wait until all messages have been fully processed before it continues; it's the sending and processing of replies from recipients which happens in parallel.
executorServiceRef		A custom Thread Pool to use for parallel processing. Note that enabling this option implies parallel processing, so you need not enable that option as well.
stopOnException	false	Whether to immediately stop processing when an exception occurs. If disabled, Camel will send the message to all recipients regardless of any individual failures. You can process exceptions in an AggregationStrategy implementation, which supports full control of error handling.
ignoreInvalidEndpoints	false	Whether to ignore an endpoint URI that could not be resolved. If disabled, Camel will throw an exception identifying the invalid endpoint URI.
streaming	false	If enabled, Camel will process replies out-of-order - that is, in the order received in reply from each recipient. If disabled, Camel will process replies in the same order as specified by the Expression.
timeout		Specifies a processing timeout milliseconds. If the Recipient List hasn't been able to send and process all replies within this timeframe, then the timeout triggers

Name	Default Value	Description
		and the Recipient List breaks out, with message flow continuing to the next element. Note that if you provide a TimeoutAwareAggregationStrategy, its {{timeout}} method is invoked before breaking out.
onPrepareRef		A custom Processor to prepare the copy of the [Exchange] each recipient will receive. This allows you to perform arbitrary transformations, such as deep-cloning the message payload (or any other custom logic).
shareUnitOfWork	false	Whether the unit of work should be shared. See the same option with the Splitter EIP for more details.

2.38.2. Static Recipient List

The following example shows how to route a request from an input **queue:a** endpoint to a static list of destinations

Using Annotations You can use the RecipientList Annotation on a POJO to create a Dynamic Recipient List. For more details see the Bean Integration.

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("mock:error"));

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

Using the Spring XML Extensions

```
<camelContext errorHandlerRef="errorHandler"
    xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <multicast>
            <to uri="seda:b"/>
            <to uri="seda:c"/>
            <to uri="seda:d"/>
        </multicast>
    </route>
</camelContext>
```

2.38.3. Dynamic Recipient List

Usually one of the main reasons for using the Recipient List pattern is that the list of recipients is dynamic and calculated at runtime. The following example demonstrates how to create a dynamic recipient list using an Expression (which in this case it extracts a named header value dynamically) to calculate the list of endpoints which are either of type Endpoint or are converted to a String and then resolved using the endpoint URIs.

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("mock:error"));

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

The above assumes that the header contains a list of endpoint URIs. The following takes a single string header and tokenizes it

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

2.38.3.1. Iteratable value

The dynamic list of recipients that are defined in the header must be iteratable such as:

- `java.util.Collection`

- `java.util.Iterator`

- arrays

- `org.w3c.dom.NodeList`

- a single String with values separated with comma

- any other type will be regarded as a single value

Using the Spring XML Extensions

```
<camelContext errorHandlerRef="errorHandler"
    xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <recipientList>
            <xpath>$foo</xpath>
        </recipientList>
    </route>
</camelContext>
```

For further examples of this pattern in use you could look at one of the junit test case

2.38.3.2. Using delimiter in Spring XML

In Spring DSL you can set the `delimiter` attribute for setting a delimiter to be used if the header value is a single String with multiple separated endpoints. By default Camel uses comma as delimiter, but this option lets you specify a customer delimiter to use instead.

```
<route>
  <from uri="direct:a" />
  <!-- use comma as a delimiter for String based values -->
```

```
    <recipientList delimiter=",">
      <header>myHeader</header>
    </recipientList>
</route>
```

So if **myHeader** contains a String with the value `"activemq:queue:foo, activemq:topic:hello ,` `log:bar"` then Camel will split the String using the delimiter given in the XML that was comma, resulting into 3 endpoints to send to. You can use spaces between the endpoints as Camel will trim the value when it lookup the endpoint to send to.

Note: In Java DSL you use the `tokenizer` to archive the same. The route above in Java DSL:

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

In **Camel 2.1** it is a bit easier as you can pass in the delimiter as second parameter:

```
from("direct:a").recipientList(header("myHeader"), "#");
```

2.38.4. Advanced Usage

2.38.4.1. Sending to multiple recipients in parallel

The Section 2.38, "Recipient List" now supports `parallelProcessing` that for example Section 2.48, "Splitter" also supports. You can use it to use a thread pool to have concurrent tasks sending the Exchange to multiple recipients concurrently.

```
from("direct:a").recipientList(header("myHeader")).parallelProcessing();
```

And in Spring XML it is an attribute on the recipient list tag.

```
<route>
      <from uri="direct:a"/>
      <recipientList parallelProcessing="true">
          <header>myHeader</header>
      </recipientList>
</route>
```

With `parallelProcessing` then you can also configure a timeout value, after which Camel will skip unprocessed messages and continue any subsequent processing. See the Camel Website for more details on this feature.

2.38.4.2. Stop continuing in case one recipient failed

The Section 2.38, "Recipient List" now supports `stopOnException` that for example Section 2.48, "Splitter" also supports. You can use it to stop sending to any further recipients in case any recipient failed.

```
from("direct:a").recipientList(header("myHeader")).stopOnException();
```

And in Spring XML it is an attribute on the recipient list tag.

```
<route>
      <from uri="direct:a"/>
      <recipientList stopOnException="true">
          <header>myHeader</header>
```

```
        </recipientList>
</route>
```

Note: You can combine `parallelProcessing` and `stopOnException` and have them both `true`.

2.38.4.3. Ignore invalid endpoints

The Section 2.38, "Recipient List" now supports `ignoreInvalidEndpoints` which the Section 2.42, "Routing Slip" also supports. You can use it to skip endpoints which is invalid.

```
from("direct:a").recipientList(header("myHeader")).
   ignoreInvalidEndpoints();
```

And in Spring XML it is an attribute on the recipient list tag.

```
<route>
        <from uri="direct:a"/>
        <recipientList ignoreInvalidEndpoints="true">
            <header>myHeader</header>
        </recipientList>
    </route>
```

Then let's say the `myHeader` contains the following two endpoints `direct:foo,xxx:bar`. The first endpoint is valid and works. However the second is invalid and will just be ignored. Camel logs at INFO level about, so you can see why the endpoint was invalid.

2.38.4.4. Using method call as recipient list

You can use a Section 3.3, "Bean" to provide the recipients, for example:

```
from("activemq:queue:test").recipientList().method(MessageRouter.class,
    "routeTo");
```

and then `MessageRouter`:

```
public class MessageRouter {

    public String routeTo() {
        String queueName = "activemq:queue:test2";
        return queueName;
    }
}
```

When you use a Section 3.3, "Bean" then do **not** also use the `@RecipientList` annotation, as this will in fact add yet another recipient list, so you end up having two. Do **not** do like this.

```
public class MessageRouter {

    @RecipientList
    public String routeTo() {
        String queueName = "activemq:queue:test2";
        return queueName;
    }
}
```

An exception to the above rule is that would use `@RecipientList` if you want to route to a Section 3.3, "Bean" which you then want to act as a recipient list. So the original route can be changed to:

```
from("activemq:queue:test").bean(MessageRouter.class, "routeTo");
```

Which then would invoke the routeTo method and detect if its annotated with `@RecipientList` and then act accordingly as if it was a recipient list EIP.

2.39. Request Reply

Camel supports the Request Reply from the EIP patterns by supporting the Exchange Pattern on a Section 2.22, "Message" which can be set to **InOut** to indicate a request/reply. Camel Components then implement this pattern using the underlying transport or protocols.

> **Note**
>
> See also the related Section 2.16, "Event Message" EIP.
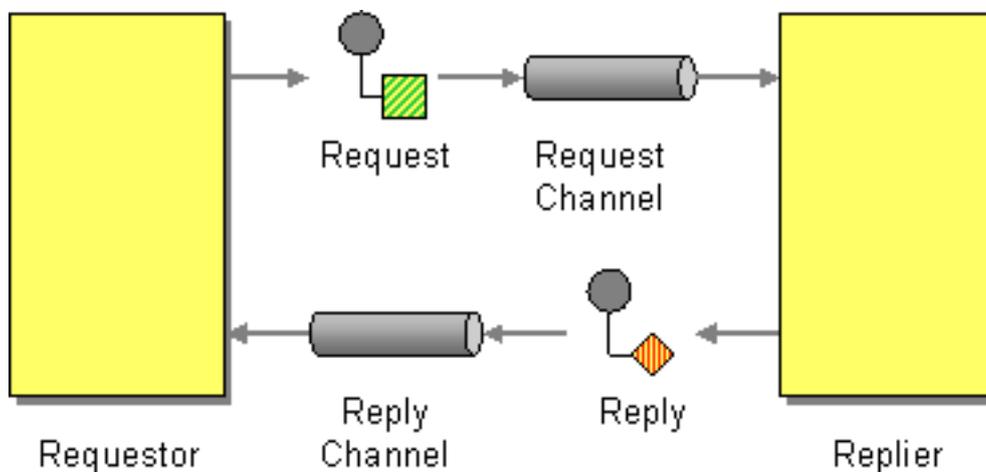
For example when using Section 3.25, "JMS" with InOut the component will by default perform these actions

- create by default a temporary inbound queue

- set the JMSReplyTo destination on the request message

- set the JMSCorrelationID on the request message

- send the request message

- consume the response and associate the inbound message to the request using the JMSCorrelationID (as you may be performing many concurrent request/responses).

2.39.1. Explicitly specifying InOut

When consuming messages from Section 3.25, "JMS" a Request-Reply is indicated by the presence of the **JMSReplyTo** header.

You can explicitly force an endpoint to be in Request Reply mode by setting the exchange pattern on the URI. e.g.

```
jms:MyQueue?exchangePattern=InOut
```

You can also specify the exchange pattern in DSL rule or Spring configuration, see the Request-Reply EIP page on the Apache Camel site for the latest updated example.

2.40. Resequencer

2.40.1. Resequencer

The Resequencer from the EIP patterns allows you to reorganise messages based on some comparator. By default in Camel we use an Expression to create the comparator; so that you can compare by a message header or the body or a piece of a message etc.
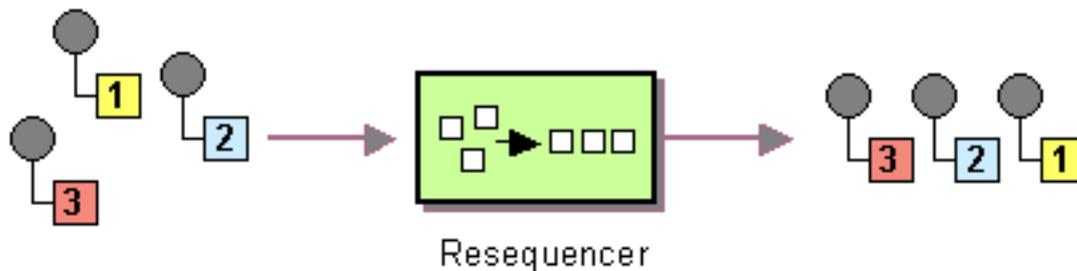
Camel supports two resequencing algorithms:

- **Batch resequencing** collects messages into a batch, sorts the messages and sends them to their output.

- **Stream resequencing** re-orders (continuous) message streams based on the detection of gaps between messages.

By default the Section 2.40, "Resequencer" does not support duplicate messages and will only keep the last message, in case a message arrives with the same message expression. However in the batch mode you can enable it to allow duplicates. For Batch mode, in Java DSL there is a `allowDuplicates()` method and in Spring XML there is an `allowDuplicates=true` attribute on the `<batch-config/>` you can use to enable it.

2.40.2. Batch Resequencing

The following example shows how to use the batch-processing resequencer so that messages are sorted in order of the **body()** expression. That is messages are collected into a batch (either by a maximum number of messages per batch or using a timeout) then they are sorted in order and then sent out to their output.

Using the Fluent Builders

```
from("direct:start")
    .resequence().body()
    .to("mock:result");
```

This is equivalent to

```
from("direct:start")
    .resequence(body()).batch()
    .to("mock:result");
```

The batch-processing resequencer can be further configured via the `size()` and `timeout()` methods.

```
from("direct:start")
    .resequence(body()).batch().size(300).timeout(4000L)
    .to("mock:result")
```

This sets the batch size to 300 and the batch timeout to 4000 ms (by default, the batch size is 100 and the timeout is 1000 ms). Alternatively, you can provide a configuration object.

```
from("direct:start")
    .resequence(body()).batch(new BatchResequencerConfig(300, 4000L))
    .to("mock:result")
```

So the above example will reorder messages from endpoint **direct:a** in order of their bodies, to the endpoint **mock:result**. Typically you'd use a header rather than the body to order things; or maybe a part of the body. So you could replace this expression with

```
resequencer(header("mySeqNo"))
```

for example to reorder messages using a custom sequence number in the header `mySeqNo`.

You can of course use many different Expression languages such as XPath, XQuery, SQL or various Scripting Languages.

Using the Spring XML Extensions

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

2.40.2.1. Reverse

In the `batch` mode, you can now reverse the expression ordering. By default the order is based on 0..9,A..Z, which would let messages with low numbers be ordered first, and thus also also outgoing first. In some cases you want to reverse order, which is now possible.

In Java DSL there is a `reverse()` method and in Spring XML there is an `reverse=true` attribute on the `<batch-config/>` you can use to enable it.

2.40.3. Stream Resequencing

The next example shows how to use the stream-processing resequencer. Messages are re-ordered based on their sequence numbers given by a `seqnum` header using gap detection and timeouts on the level of individual messages.

Using the Fluent Builders

```
from("direct:start").resequence(header("seqnum")).
   stream().to("mock:result");
```

The stream-processing resequencer can be further configured via the `capacity()` and `timeout()` methods.

```
from("direct:start")
    .resequence(header("seqnum")).stream().capacity(5000).timeout(4000L)
    .to("mock:result")
```

This sets the resequencer's capacity to 5000 and the timeout to 4000 ms (by default, the capacity is 1000 and the timeout is 1000 ms). Alternatively, you can provide a configuration object.

```
from("direct:start")
    .resequence(header("seqnum")).stream(
    new StreamResequencerConfig(5000, 4000L)).to("mock:result")
```

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

If the maximum time difference between messages (with successor/predecessor relationship with respect to the sequence number) in a message stream is known, then the resequencer's timeout parameter should be set to this value. In this case it is guaranteed that all messages of a stream are delivered in correct order to the next processor. The lower the timeout value is compared to the out-of-sequence time difference the higher is the probability for out-of-sequence messages delivered by this resequencer. Large timeout values should be supported by sufficiently high capacity values. The capacity parameter is used to prevent the resequencer from running out of memory.

By default, the stream resequencer expects `long` sequence numbers but other sequence numbers types can be supported as well by providing a custom expression.

```
public class MyFileNameExpression implements Expression {

    public String getFileName(Exchange exchange) {
        return exchange.getIn().getBody(String.class);
    }

    public Object evaluate(Exchange exchange) {
        // parse the file name with YYYYMMDD-DNNN pattern
        String fileName = getFileName(exchange);
        String[] files = fileName.split("-D");
        Long answer = Long.parseLong(files[0]) * 1000 +
            Long.parseLong(files[1]);
        return answer;
    }


    public <T> T evaluate(Exchange exchange, Class<T> type) {
        Object result = evaluate(exchange);
        return exchange.getContext().getTypeConverter().convertTo(type,
            result);
    }

}
```

or custom comparator via the `comparator()` method

```
ExpressionResultComparator<Exchange> comparator = new MyComparator();
from("direct:start")
    .resequence(header("seqnum")).stream().comparator(comparator)
    .to("mock:result");
```

or via a `StreamResequencerConfig` object.

```
ExpressionResultComparator<Exchange> comparator = new MyComparator();
StreamResequencerConfig config = new StreamResequencerConfig(100, 1000L,
```

```
comparator);

from("direct:start")
    .resequence(header("seqnum")).stream(config)
    .to("mock:result");
```

Using the Spring XML Extensions

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

2.40.4. Further Examples

See the Camel Website for the further examples of this component in use.

2.41. Return Address

Camel supports the Return Address from the EIP patterns by using the `JMSReplyTo` header.
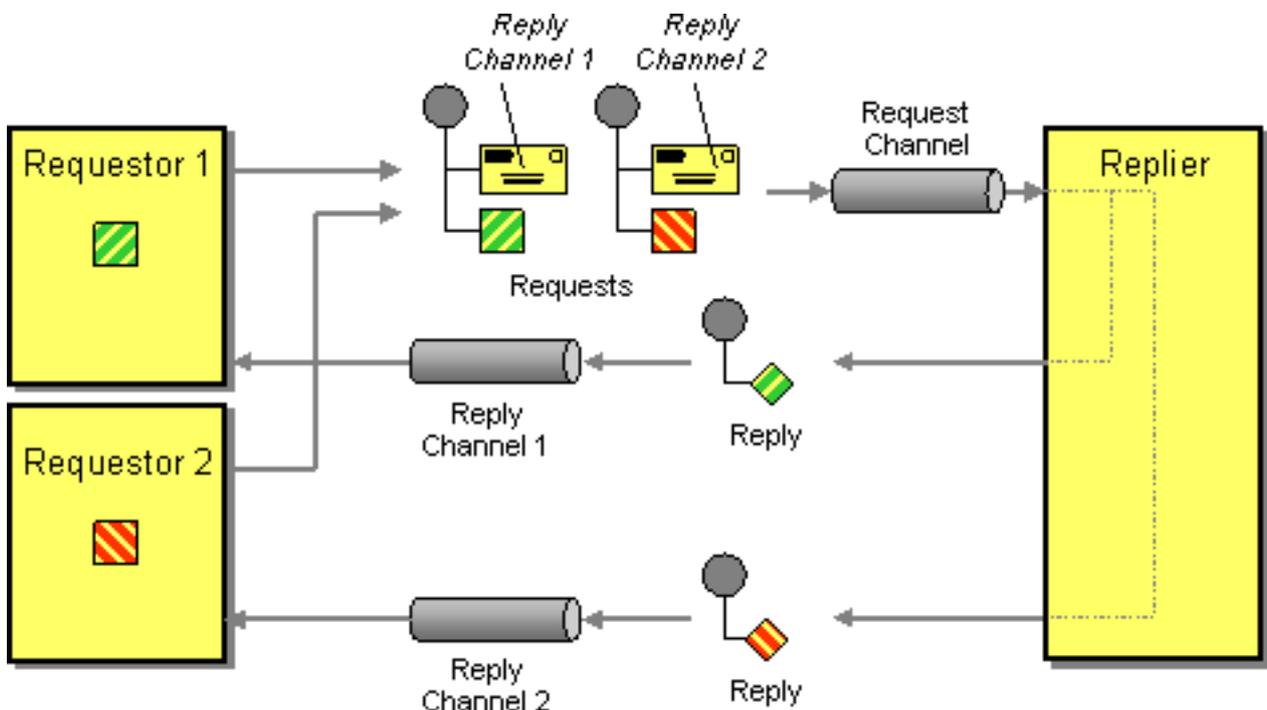
For example when using Section 3.25, "JMS" with InOut the component will by default return to the address given in `JMSReplyTo`.

___

Requestor Code:

```
getMockEndpoint("mock:bar").expectedBodiesReceived("Bye World");
template.sendBodyAndHeader("direct:start", "World", "JMSReplyTo",
    "queue:bar");
```

Route Using the Fluent Builders:

```
from("direct:start").to("activemq:queue:foo?preserveMessageQos=true");
from("activemq:queue:foo").transform(body().prepend("Bye "));
from("activemq:queue:bar?disableReplyTo=true").to("mock:bar");
```

Route Using the Spring XML Extensions:

```
<route>
<from uri="direct:start"/>
<to uri="activemq:queue:foo?preserveMessageQos=true"/>
</route>

<route>
<from uri="activemq:queue:foo"/>
<transform>
<simple>Bye ${in.body}</simple>
</transform>
</route>
```

<route> <from uri="activemq:queue:bar?disableReplyTo=true"/> <to uri="mock:bar"/> </route> {code}

For a complete example of this pattern, see this JUnit test case

2.42. Routing Slip

2.42.1. Routing Slip

The Routing Slip from the EIP patterns allows you to route a message consecutively through a series of processing steps where the sequence of steps is not known at design time and can vary for each message.

Attach Routing Slip
to Message

Route Message
According to Slip
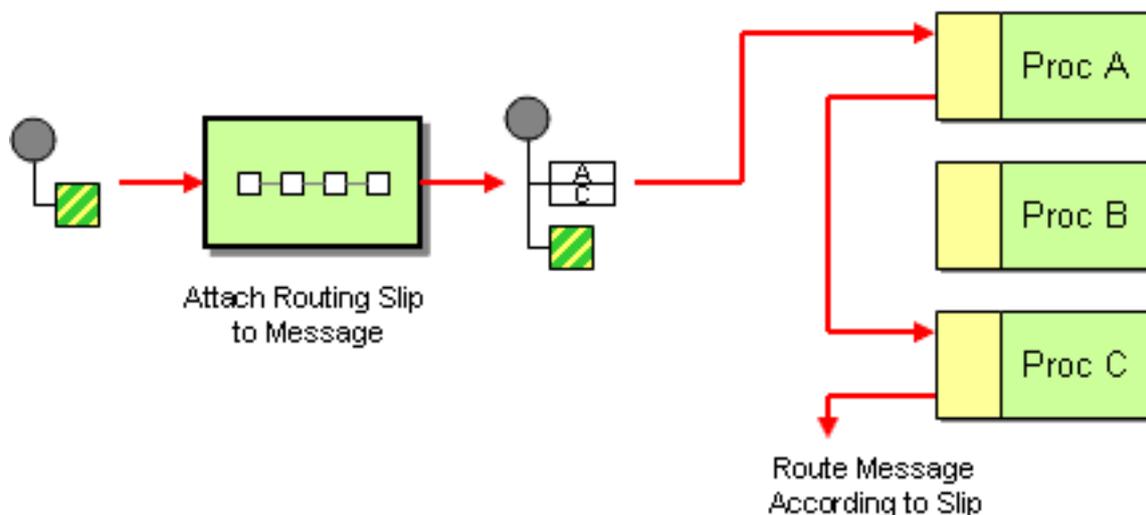
2.42.1.1. Example

The following route will take any messages sent to the Apache ActiveMQ queue **SomeQueue** and pass them into the Routing Slip pattern.

```
from("activemq:SomeQueue").routingSlip("headerName");
```

Messages will be checked for the existance of the "headerName" header. The value of this header should be a comma-delimited list of endpoint URIs you wish the message to be routed to. The Section 2.22, "Message" will be routed in a pipeline fashion (i.e. one after the other).

From Camel 2.5 the Section 2.42, "Routing Slip" will set a property (`Exchange.SLIP_ENDPOINT`) on the Exchange which contains the current endpoint as it advanced though the slip. This allows you to *know* how far we have processed in the slip.

The Section 2.42, "Routing Slip" will compute the slip **beforehand** which means, the slip is only computed once. If you need to compute the slip *on-the-fly* then use the Section 2.14, "Dynamic Router" pattern instead.

2.42.1.2. Configuration options

Here we set the header name and the URI delimiter to something different.

Using the Fluent Builders

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

Using the Spring XML Extensions

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

2.42.2. Ignore invalid endpoints

The Section 2.42, "Routing Slip" now supports `ignoreInvalidEndpoints` which the Section 2.38, "Recipient List" also supports. You can use it to skip endpoints which is invalid.

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

And in Spring XML it is an attribute on the recipient list tag.

```
<route>
      <from uri="direct:a"/>
      <routingSlip headerName="myHeader" ignoreInvalidEndpoints="true"/>
</route>
```

Then let's say the `myHeader` contains the following two endpoints `direct:foo,xxx:bar`. The first endpoint is valid and works. However the second is invalid and will just be ignored. Camel logs at INFO level about, so you can see why the endpoint was invalid.

2.42.3. Expression supporting

The Section 2.42, "Routing Slip" now supports to take the expression parameter as the Section 2.38, "Recipient List" does. You can tell the camel the expression that you want to use to get the routing slip.

```
from("direct:a").routingSlip(header("myHeader")).ignoreInvalidEndpoints();
```

And in Spring XML it is an attribute on the recipient list tag.

```
<route>
        <from uri="direct:a"/>
        <!--NOTE you need to specify the expression element
            inside of the routingSlip element -->
        <routingSlip ignoreInvalidEndpoints="true">
            <header>myHeader</header>
        </routingSlip>
</route>
```

2.42.3.1. Further Examples

For further examples of this pattern in use you could look at the routing slip test cases.

2.43. Sampling

A sampling throttler allows you to extract a sample of the exchanges from the traffic through a route. It is configured with a sampling period during which only a single exchange is allowed to pass through. All other exchanges will be stopped.

Will by default use a sample period of 1 second. Options:

Name	Default Value	Description
messageFrequency	(none)	Samples the message every N'th message. You can use either frequency or period.
samplePeriod	1	Samples the message every N'th message. You can use either frequency or period.
units	seconds	Time unit as an enum of java.util.concurrent.TimeUnit from the JDK.

2.43.1. Samples

You use this EIP with the `sample` DSL as show in these samples.

Using the Fluent Builders These samples also show how you can use the different syntax to configure the sampling period:

```
from("direct:sample")
```

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

from("direct:sample-configured")
    .sample(1, TimeUnit.SECONDS)
    .to("mock:result");

from("direct:sample-configured-via-dsl")
    .sample().samplePeriod(1).timeUnits(TimeUnit.SECONDS)
    .to("mock:result");

from("direct:sample-messageFrequency")
    .sample(10)
    .to("mock:result");

from("direct:sample-messageFrequency-via-dsl")
    .sample().sampleMessageFrequency(5)
    .to("mock:result");
```

Using the Spring XML Extensions And the same example in Spring XML is:

```
<route>
    <from uri="direct:sample"/>
    <sample samplePeriod="1" units="seconds">
        <to uri="mock:result"/>
    </sample>
</route>
<route>
    <from uri="direct:sample-messageFrequency"/>
    <sample messageFrequency="10">
        <to uri="mock:result"/>
    </sample>
</route>
<route>
    <from uri="direct:sample-messageFrequency-via-dsl"/>
    <sample messageFrequency="5">
        <to uri="mock:result"/>
    </sample>
</route>
```

And since it uses a default of 1 second you can omit this configuration in case you also want to use 1 second

```
<route>
    <from uri="direct:sample"/>
    <!-- will by default use 1 second period -->
    <sample>
        <to uri="mock:result"/>
    </sample>
</route>
```

2.44. Scatter-Gather

The Scatter-Gather from the EIP patterns allows you to route messages to a number of dynamically specified recipients and re-aggregate the responses back into a single message.

___
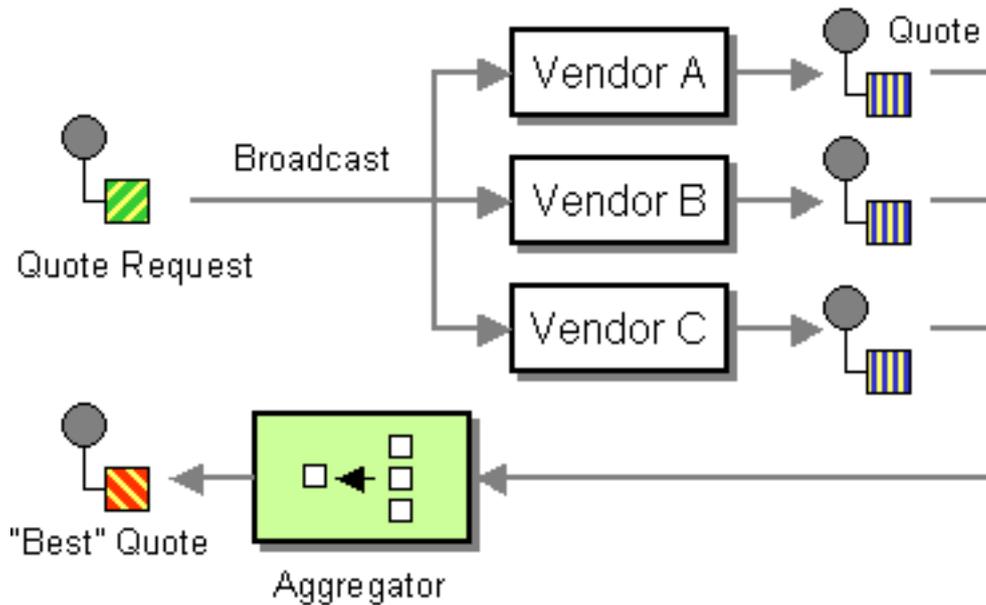
2.44.1. Dynamic Scatter-Gather Example

In this example we want to get the best quote for beer from several different vendors. We use a dynamic Section 2.38, "Recipient List" to get the request for a quote to all vendors and an Section 2.2, "Aggregator" to pick the best quote out of all the responses. The routes for this are defined as:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <recipientList>
      <header>listOfVendors</header>
    </recipientList>
  </route>
  <route>
    <from uri="seda:quoteAggregator"/>
    <aggregate strategyRef="aggregatorStrategy" completionTimeout="1000">
      <correlationExpression>
        <header>quoteRequestId</header>
      </correlationExpression>
      <to uri="mock:result"/>
    </aggregate>
  </route>
</camelContext>
```

So in the first route you see that the Section 2.38, "Recipient List" is looking at the `listOfVendors` header for the list of recipients. So, we need to send a message like

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("listOfVendors", "bean:vendor1, bean:vendor2, bean:vendor3");
headers.put("quoteRequestId", "quoteRequest-1");
template.sendBodyAndHeaders("direct:start", "<quote_request item=\"beer\"/>",
headers);
```

This message will be distributed to the following Endpoint s: `bean:vendor1`, `bean:vendor2`, and `bean:vendor3`. These are all beans which look like

```
public class MyVendor {
```

```
    private int beerPrice;

    @Produce(uri = "seda:quoteAggregator")
    private ProducerTemplate quoteAggregator;

    public MyVendor(int beerPrice) {
        this.beerPrice = beerPrice;
    }

    public void getQuote(@XPath("/quote_request/@item") String item,
        Exchange exchange) throws Exception {
        if ("beer".equals(item)) {
            exchange.getIn().setBody(beerPrice);
            quoteAggregator.send(exchange);
        } else {
            throw new Exception("No quote available for " + item);
        }
    }
}
```

and are loaded up in Spring like

```
<bean id="aggregatorStrategy" class=
    "org.apache.camel.spring.processor.scattergather. \\
     LowestQuoteAggregationStrategy"/>


<bean id="vendor1"
    class="org.apache.camel.spring.processor.scattergather.MyVendor">
  <constructor-arg>
    <value>1</value>
  </constructor-arg>
</bean>


<bean id="vendor2"
    class="org.apache.camel.spring.processor.scattergather.MyVendor">
  <constructor-arg>
    <value>2</value>
  </constructor-arg>
</bean>


<bean id="vendor3"
    class="org.apache.camel.spring.processor.scattergather.MyVendor">
  <constructor-arg>
    <value>3</value>
  </constructor-arg>
</bean>
```

Each bean is loaded with a different price for beer. When the message is sent to each bean endpoint, it will arrive at the `MyVendor.getQuote` method. This method does a simple check whether this quote request is for beer and then sets the price of beer on the exchange for retrieval at a later step. The message is forwarded on to the next step using POJO Producing (see the @Produce annotation).

At the next step we want to take the beer quotes from all vendors and find out which one was the best (i.e. the lowest!). To do this we use an Section 2.2, "Aggregator" with a custom aggregation strategy. The Section 2.2, "Aggregator" needs to be able to compare only the messages from this particular quote; this is easily done by specifying a correlationExpression equal to the value of the quoteRequestId header. As shown above in the message sending snippet, we set this header to `quoteRequest-1`. This correlation value should be unique or you may include responses that are not part of this quote. To pick the lowest quote out of the set, we use a custom aggregation strategy like

```
public class LowestQuoteAggregationStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        // the first time we only have the new exchange
        if (oldExchange == null) {
            return newExchange;
        }

        if (oldExchange.getIn().getBody(int.class)
            < newExchange.getIn().getBody(int.class)) {
            return oldExchange;
        } else {
            return newExchange;
        }
    }
}
```

Finally, we expect to get the lowest quote of $1 out of $1, $2, and $3.

```
result.expectedBodiesReceived(1); // expect the lowest quote
```

You can find the full example source here:

camel-spring/src/test/java/org/apache/camel/spring/processor/scattergather/

camel-spring/src/test/resources/org/apache/camel/spring/processor/scattergather/scatter-gather.xml

2.44.2. Static Scatter-Gather Example

You can lock down which recipients are used in the Scatter-Gather by using a static Section 2.38, "Recipient List".
It looks something like this

```
from("direct:start").multicast().to("seda:vendor1", "seda:vendor2",
    "seda:vendor3");

from("seda:vendor1").to("bean:vendor1").to("seda:quoteAggregator");
from("seda:vendor2").to("bean:vendor2").to("seda:quoteAggregator");
from("seda:vendor3").to("bean:vendor3").to("seda:quoteAggregator");

from("seda:quoteAggregator")
    .aggregate(header("quoteRequestId"), new LowestQuoteAggregationStrategy()).to(
    "mock:result")
```

2.45. Selective Consumer

The Selective Consumer from the EIP patterns can be implemented in two ways
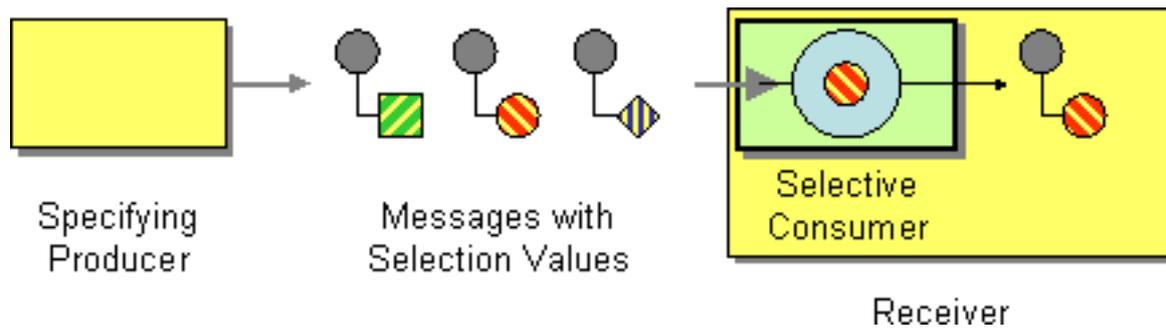
The first solution is to provide a Message Selector to the underlying URIs when creating your consumer. For example when using Section 3.25, "JMS" you can specify a selector parameter so that the message broker will only deliver messages matching your criteria.

The other approach is to use a Section 2.27, "Message Filter" which is applied; then if the filter matches the message your consumer is invoked as shown in the following example

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("mock:error"));

        from("seda:a")
            .filter(header("foo").isEqualTo("bar"))
                .process(myProcessor);
    }
};
```

Using the Spring XML Extensions

```
<bean id="myProcessor" class="org.apache.camel.builder.MyProcessor"/>

<camelContext errorHandlerRef="errorHandler"
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <filter>
            <xpath>$foo = 'bar'</xpath>
            <process ref="myProcessor"/>
        </filter>
    </route>
</camelContext>
```

2.46. Service Activator

Camel has several endpoint components that support the Service Activator from the EIP patterns.
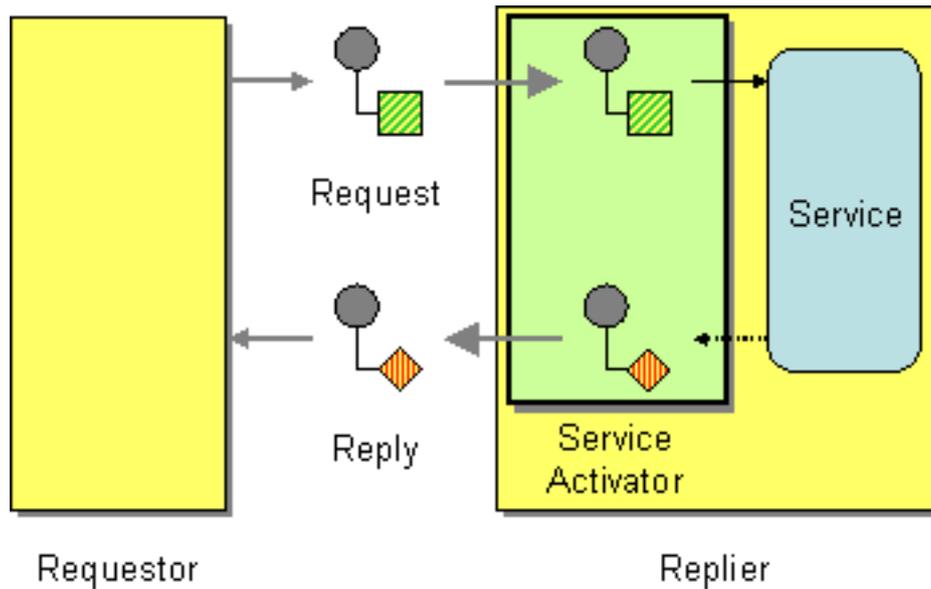
Components like Section 3.3, "Bean", Section 3.8, "CXF" and Pojo provide a a way to bind the message exchange to a Java interface/service where the route defines the endpoints and wires it up to the bean.

In addition you can use the Bean Integration to wire messages to a bean using annotation.

Here is a simple example of using a Direct endpoint to create a messaging interface to a Pojo Bean service. Using the Fluent Builders:

```
from("direct:invokeMyService").to("bean:myService");
```

Using the Spring XML Extensions:

```
<route>
    <from uri="direct:invokeMyService"/>
    <to uri="bean:myService"/>
</route>
```

2.47. Sort

Sort can be used to sort a message. Imagine you consume text files and before processing each file you want to be sure the content is sorted.

Sort will by default sort the body using a default comparator that handles numeric values or uses the string representation. You can provide your own comparator, and even an expression to return the value to be sorted. Sort requires the value returned from the expression evaluation is convertible to `java.util.List` as this is required by the JDK sort operation.

Name	Default Value	Description
comparatorRef	A->Z sorting	Refers to a custom java.util.Comparator to use for sorting the message body. Camel will by default use a comparator which does a A..Z sorting.

2.47.1. Using from Java DSL

In the route below it will read the file content and tokenize by line breaks so each line can be sorted.

```
from("file://inbox").sort(body().tokenize("\n")).to(
"bean:MyServiceBean.processLine");
```

You can pass in your own comparator as a second argument:

```
from("file://inbox").sort(body().tokenize("\n"), new MyReverseComparator())
   .to("bean:MyServiceBean.processLine");
```

2.47.2. Using from Spring DSL

In the route below it will read the file content and tokenize by line breaks so each line can be sorted.

Example 2.1. Camel 2.7 onwards

```
<route>
  <from uri="file://inbox"/>
  <sort>
    <simple>body</simple>
  </sort>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>
```

Example 2.2. Camel 2.6 or older

```
<route>
  <from uri="file://inbox"/>
  <sort>
    <expression>
      <simple>body</simple>
    </expression>
  </sort>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>
```

And to use our own comparator we can refer to it as a Spring bean:

Example 2.3. Camel 2.7 or older

```
<route>
  <from uri="file://inbox"/>
  <sort comparatorRef="myReverseComparator">
    <simple>body</simple>
  </sort>
  <beanRef ref="MyServiceBean" method="processLine"/>
</route>

<bean id="myReverseComparator" class="com.mycompany.MyReverseComparator"/>
```

Example 2.4. Camel 2.6 or older

```
<route>
  <from uri="file://inbox"/>
  <sort comparatorRef="myReverseComparator">
    <expression>
      <simple>body</simple>
    </expression>
  </sort>
  <beanRef ref="MyServiceBean" method="processLine"/>
</route>

<bean id="myReverseComparator" class="com.mycompany.MyReverseComparator"/>
```

Besides `<simple>`, you can supply an expression using any language you like, so long as it returns a list.

2.48. Splitter

2.48.1. Splitter

The Splitter from the EIP patterns allows you split a message into a number of pieces and process them individually
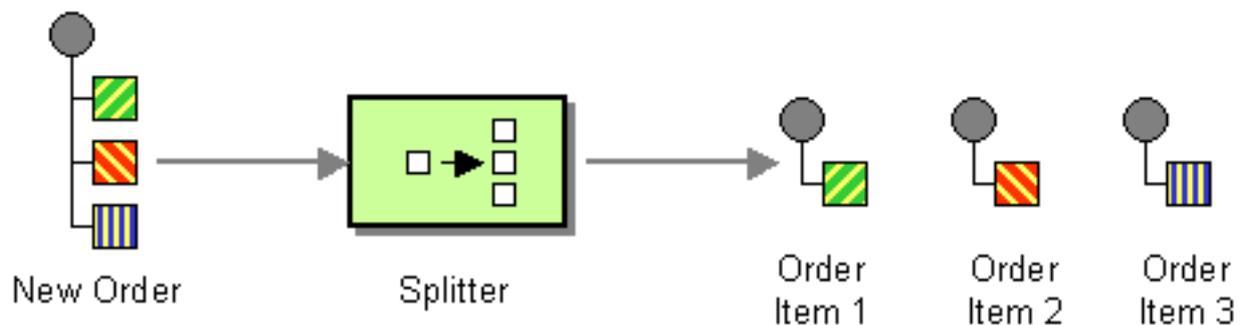
You need to specify a Splitter as `split()`. In earlier versions of Camel, you need to use `splitter()`.

Options:

Name	Default Value	Description
strategyRef		Refers to an AggregationStrategy to be used to assemble the replies from the sub-messages, into a single outgoing message from the Splitter. See the defaults described below in What the Splitter returns.
parallelProcessing	false	If enables then processing the sub-messages occurs concurrently. Note the caller thread will still wait until all sub-messages has been fully processed, before it continues.
executorServiceRef		Refers to a custom Thread Pool to be used for parallel processing. Notice if you set this option, then parallel processing is automatic implied, and you do not have to enable that option as well.
stopOnException	false	Camel 2.2: Whether or not to stop continue processing immediately when an exception occurred. If disable,

Name	Default Value	Description
		then Camel continue splitting and process the sub-messages regardless if one of them failed. You can deal with exceptions in the AggregationStrategy class where you have full control how to handle that.
streaming	false	If enabled then Camel will split in a streaming fashion, which means it will split the input message in chunks. This reduces the memory overhead. For example if you split big messages its recommended to enable streaming. If streaming is enabled then the sub-message replies will be aggregated out-of-order, eg in the order they come back. If disabled, Camel will process sub-message replies in the same order as they where splitted.
timeout		Camel 2.5: Sets a total timeout specified in millis. If the Recipient List hasn't been able to split and process all replies within the given timeframe, then the timeout triggers and the Splitter breaks out and continues. Notice if you provide a TimeoutAwareAggregationStrategy then the timeout method is invoked before breaking out.
onPrepareRef		Camel 2.8: Refers to a custom Processor to prepare the sub-message of the Exchange, before its processed. This allows you to do any custom logic, such as deep-cloning the message payload if that's needed etc.
shareUnitOfWork	false	Camel 2.8: Whether the unit of work should be shared. See further below for more details.

Exchange Properties:

Property	Type	Description
CamelSplitIndex	int	Camel 2.0: A split counter that increases for each Exchange being split. The counter starts from 0.
CamelSplitSize	int	Camel 2.0: The total number of Exchanges that was splitted. This header is not applied for stream based splitting. From Camel 2.9 onwards this header is also set in stream based splitting, but only on the completed Exchange.
CamelSplitComplete	boolean	Whether or not this Exchange is the last.

What does the splitter return?

The Section 2.48, "Splitter" will by default return the **last** splitted message.

The Section 2.48, "Splitter" will by default return the original input message.

For all versions You can override this by suppling your own strategy as an `AggregationStrategy`. There is a sample on this page (Split aggregate request/reply sample). Notice it is the same strategy as the Section 2.2, "Aggregator" supports. This Section 2.48, "Splitter" can be viewed as having a build in light weight Section 2.2, "Aggregator".

2.48.1.1. Example

The following example shows how to take a request from the **queue:a** endpoint the split it into pieces using an Expression, then forward each piece to **queue:b**

Using the Fluent Builders

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        errorHandler(deadLetterChannel("mock:error"));

        from("seda:a")
            .split(body(String.class).tokenize("\n"))
                .to("seda:b");
    }
};
```

The splitter can use any Expression language so you could use any of the Languages Supported such as XPath, XQuery, SQL or one of the Scripting Languages to perform the split. e.g.

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

Using the Spring XML Extensions

```
<camelContext errorHandlerRef="errorHandler"
    xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:a"/>
        <split>
            <xpath>/invoice/lineItems</xpath>
            <to uri="seda:b"/>
        </split>
    </route>
</camelContext>
```

For further examples of this pattern in use you could look at one of the junit test case

Using Tokenizer from Spring XML Extensions

You can use the tokenizer expression in the Spring DSL to split bodies or headers using a token. This is a common use-case, so we provided a special **tokenizer** tag for this. In the sample below we split the body using a @ as separator. You can of course use comma or space or even a regex pattern, also set regex=true.

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <split>
            <tokenize token="@"/>
            <to uri="mock:result"/>
        </split>
    </route>
</camelContext>
```

Splitting the body in Spring XML is a bit harder as you need to use the Simple language to dictate this

```
<split>
   <simple>${body}</simple>
   <to uri="mock:result"/>
</split>
```

2.48.1.2. Exchange properties

The following properties is set on each Exchange that is split:

header	type	description
`CamelSplitIndex`	int	A split counter that increases for each Exchange being split. The counter starts from 0.
`CamelSplitSize`	int	The total number of Exchanges that was splitted. This header is not applied for stream based splitting.
`CamelSplitComplete`	boolean	Whether or not this Exchange is the last.

2.48.1.3. Parallel execution of distinct 'parts'

If you want to execute all parts in parallel you can use special notation of `split()` with two arguments, where the second one is a **boolean** flag if processing should be parallel. e.g.

```
XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");
from("activemq:my.queue").split(xPathBuilder, true).to(
    "activemq:my.parts");
```

In the boolean option has been refactored into a builder method `parallelProcessing` so it is easier to understand what the route does when we use a method instead of true|false.

```
XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");
from("activemq:my.queue").split(xPathBuilder).parallelProcessing().
    to("activemq:my.parts");
```

2.48.1.4. Stream based

The XPath engine in Java and XQuery will load the entire XML content into memory. And thus they are not well suited for very big XML payloads. Instead you can use a custom Expression which will iterate the XML payload in a streamed fashion. From Camel 2.9 onwards you can use the Tokenizer language which supports this when you supply the start and end tokens.

You can split streams by enabling the streaming mode using the `streaming` builder method.

```
from("direct:streaming").split(body().tokenize(",")).streaming().
    to("activemq:my.parts");
```

You can also supply your custom splitter to use with streaming like this:

```
import static org.apache.camel.builder.ExpressionBuilder.beanExpression;
from("direct:streaming")
    .split(beanExpression(new MyCustomIteratorFactory(),  "iterator"))
    .streaming().to("activemq:my.parts")
```

2.48.1.5. Streaming big XML payloads using Tokenizer language

If you have a big XML payload, from a file source, and want to split it in streaming mode, then you can use the Tokenizer language with start/end tokens to do this with low memory footprint. (Note the Camel StAX component can also be used to split big XML files in a streaming mode.) See the Camel Website for an example.

2.48.1.6. Specifying a custom aggregation strategy

This is specified similar to the Section 2.2, "Aggregator".

2.48.1.7. Specifying a custom ThreadPoolExecutor

You can customize the underlying ThreadPoolExecutor used in the parallel splitter. In the Java DSL try something like this:

```
XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");

ExecutorService pool = ...

from("activemq:my.queue")
    .split(xPathBuilder).parallelProcessing().executorService(pool)
        .to("activemq:my.parts");
```

2.48.1.8. Using a Pojo to do the splitting

As the Section 2.48, "Splitter" can use any Expression to do the actual splitting we leverage this fact and use a **method** expression to invoke a Section 3.3, "Bean" to get the splitted parts. The Section 3.3, "Bean" should return a value that is iterable such as: java.util.Collection, java.util.Iterator or an array.

In the route we define the Expression as a method call to invoke our Section 3.3, "Bean" that we have registered with the id mySplitterBean in the Registry.

```
from("direct:body")
    // here we use a POJO bean mySplitterBean to do the split of the payload
    .split().method("mySplitterBean", "splitBody")
    .to("mock:result");
from("direct:message")
    // here we use a POJO bean mySplitterBean to do the split of the message
    // with a certain header value
    .split().method("mySplitterBean", "splitMessage")
    .to("mock:result");
```

And the logic for our Section 3.3, "Bean" is as simple as. Notice we use Camel Bean Binding to pass in the message body as a String object.

```
public class MySplitterBean {

    /**
     * The split body method returns something that is iteratable
     * such as a java.util.List.
     *
     * @param body the payload of the incoming message
     * @return a list containing each part splitted
     */
    public List<String> splitBody(String body) {
        // since this is based on an unit test you can of cause
        // use different logic for splitting as Camel have out
        // of the box support for splitting a String based on comma
        // but this is for show and tell, since this is Java code
        // you have the full power how you like to split your messages
        List<String> answer = new ArrayList<String>();
        String[] parts = body.split(",");
        for (String part : parts) {
            answer.add(part);
        }
        return answer;
```

```
    }

    /**
     * The split message method returns something that is iterable
     * such as a java.util.List.
     *
     * @param header the header of the incoming message with the name user
     * @param body the payload of the incoming message
     * @return a list containing each part splitted
     */
    public List<Message> splitMessage(@Header(value = "user")
        String header, @Body String body) {
        // we can leverage the Parameter Binding Annotations
        // http://camel.apache.org/parameter-binding-annotations.html
        // to access the message header and body at same time,
        // then create the message that we want, splitter will
        // take care rest of them.
        // *NOTE* this feature requires Camel version >= 1.6.1
        List<Message> answer = new ArrayList<Message>();
        String[] parts = header.split(",");
        for (String part : parts) {
            DefaultMessage message = new DefaultMessage();
            message.setHeader("user", part);
            message.setBody(body);
            answer.add(message);
        }
        return answer;
    }
}
```

2.48.1.9. Split aggregate request/reply sample

This sample shows how you can split an Exchange, process each splitted message, aggregate and return a combined response to the original caller using request/reply.

The route below illustrates this and how the split supports a **aggregationStrategy** to hold the in progress processed messages:

```
// this routes starts from the direct:start endpoint
// the body is then splitted based on @ separator
// the splitter in Camel supports InOut as well and for that we need
// to be able to aggregate what response we need to send back, so we
// provide our own strategy with the class MyOrderStrategy.
from("direct:start")
    .split(body().tokenize("@"), new MyOrderStrategy())
        // each splitted message is then sent to this bean
        // where we can process it
        .to("bean:MyOrderService?method=handleOrder")
        // this is important to end the splitter route as we do not
        // want to do more routing
        // on each splitted message
    .end()
    // after we have splitted and handled each message we want to
    // send a single combined response back to the original caller,
    // so we let this bean build it for us
    // this bean will receive the result of the aggregate strategy:
```

```
            // MyOrderStrategy
            .to("bean:MyOrderService?method=buildCombinedResponse")
```

And the OrderService bean is as follows:

```
// this routes starts from the direct:start endpoint
// the body is then splitted based on @ separator
// the splitter in Camel supports InOut as well and for that we need
// to be able to aggregate what response we need to send back, so we
// provide our own strategy with the class MyOrderStrategy.
from("direct:start")
    .split(body().tokenize("@"), new MyOrderStrategy())
        // each splitted message is then send to this bean where we can
        // process it
        .to("bean:MyOrderService?method=handleOrder")
        // this is important to end the splitter route as we do not
        // want to do more routing on each splitted message
    .end()
    // after we have splitted and handled each message we want to send a
    // single combined response back to the original caller,
    // so we let this bean build it for us.  This bean will receive
    // the result of the aggregate strategy: MyOrderStrategy
    .to("bean:MyOrderService?method=buildCombinedResponse")
```

And our custom **aggregationStrategy** that is responsible for holding the in progress aggregated message that after the splitter is ended will be sent to the **buildCombinedResponse** method for final processing before the combined response can be returned to the waiting caller.

```
/**
 * This is our own order aggregation strategy where we can control
 * how each splitted message should be combined. As we do not want to
 * loos any message we copy from the new to the old to preserve the
 * order lines as long we process them
 */
public static class MyOrderStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        // put order together in old exchange
        // by adding the order from new exchange

        if (oldExchange == null) {
            // the first time we aggregate we only have the new exchange,
            // so we just return it
            return newExchange;
        }

        String orders = oldExchange.getIn().getBody(String.class);
        String newLine = newExchange.getIn().getBody(String.class);

        LOG.debug("Aggregate old orders: " + orders);
        LOG.debug("Aggregate new order: " + newLine);

        // put orders together separating by semi colon
        orders = orders + ";" + newLine;
        // put combined order back on old to preserve it
        oldExchange.getIn().setBody(orders);

        // return old as this is the one that has all
        // the orders gathered until now
```

```
        return oldExchange;
    }
}
```

So let's run the sample and see how it works. We send an Exchange to the **direct:start** endpoint containing a IN body with the String value: A@B@C. The flow is:

```
HandleOrder: A
HandleOrder: B
Aggregate old orders: (id=1,item=A)
Aggregate new order: (id=2,item=B)
HandleOrder: C
Aggregate old orders: (id=1,item=A);(id=2,item=B)
Aggregate new order: (id=3,item=C)
BuildCombinedResponse: (id=1,item=A);(id=2,item=B);(id=3,item=C)
Response to caller: Response[(id=1,item=A);(id=2,item=B);(id=3,item=C)]
```

2.48.2. Stop processing in case of exception

The Section 2.48, "Splitter" will by default continue to process the entire Exchange even in case of one of the splitted message will thrown an exception during routing. For example if you have an Exchange with 1000 rows that you split and route each sub message. During processing of these sub messages an exception is thrown at the 17th. What Camel does by default is to process the remainder 983 messages. You have the chance to remedy or handle this in the AggregationStrategy.

But sometimes you just want Camel to stop and let the exception be propagated back, and let the Camel error handler handle it. You can do this by specifying that it should stop in case of an exception occurred. This is done by the stopOnException option as shown below:

```
from("direct:start")
        .split(body().tokenize(",")).stopOnException()
            .process(new MyProcessor())
            .to("mock:split");
```

And using XML DSL you specify it as follows:

```
<route>
            <from uri="direct:start"/>
            <split stopOnException="true">
                <tokenize token=","/>
                <process ref="myProcessor"/>
                <to uri="mock:split"/>
            </split>
        </route>
```

2.48.3. Sharing Unit of Work

The Splitter will by default not share a unit of work between the parent exchange and each splitted exchange. This means each sub exchange has its own individual unit of work. For example you may have an use case, where you want to split a big message, and you want to regard that process as an atomic isolated operation that either is a success or failure. In case of a failure you want that big message to be moved into a dead letter queue. To support this use case, you would have to share the unit of work on the Splitter. See the online example maintained on the Apache Camel site for more information.

```
XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");
from("activemq:my.queue").split(xPathBuilder).parallelProcessing().
    to("activemq:my.parts");
```

2.49. Throttler

2.49.1. Throttler

The Throttler Pattern allows you to ensure that a specific endpoint does not get overloaded, or that we don't exceed an agreed SLA with some external service.

Options:

Name	Default Value	Description
maximumRequestsPerPeriod		Maximum number of requests per period to throttle. This option must be provided and a positive number. Note, in the XML DSL, from Camel 2.8 onwards this option is configured using an Expression instead of an attribute.
timePeriodMillis	1000	The time period in millis, in which the throttler will allow at most maximumRequestsPerPeriod number of messages.
asyncDelayed	false	If enabled then any messages which is delayed happens asynchronously using a scheduled thread pool.
executorServiceRef		Refers to a custom Thread Pool to be used if asyncDelay has been enabled.
callerRunsWhenRejected	true	Is used if asyncDelayed was enabled. This controls if the caller thread should execute the task if the thread pool rejected the task.

Using the Fluent Builders

```
from("seda:a").throttle(3).timePeriodMillis(10000).to("log:result",
    "mock:result");
```

The above example will throttle messages all messages received on **seda:a** before being sent to **mock:result** ensuring that a maximum of 3 messages are sent in any 10 second window. Note that typically you would often use the default time period of a second. So to throttle requests at 100 requests per second between two endpoints it would look more like this...

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

For further examples of this pattern in use you could look at the junit test case

Using the Spring XML Extensions

```
<route>
  <from uri="seda:a" />
  <throttle timePeriodMillis="10000"/>
    <constant>3</constant>
    <to uri="mock:result" />
  </throttle>
```

```
</route>
```

2.49.2. Asynchronous delaying

You can let the Section 2.49, "Throttler" use non blocking asynchronous delaying, which means Camel will use a scheduler to schedule a task to be executed in the future. The task will then continue routing. This allows the caller thread to not block and be able to service other messages etc.

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

2.50. Transactional Client

2.50.1. Transactional Client

Camel recommends supporting the Transactional Client from the EIP patterns using Spring transactions.
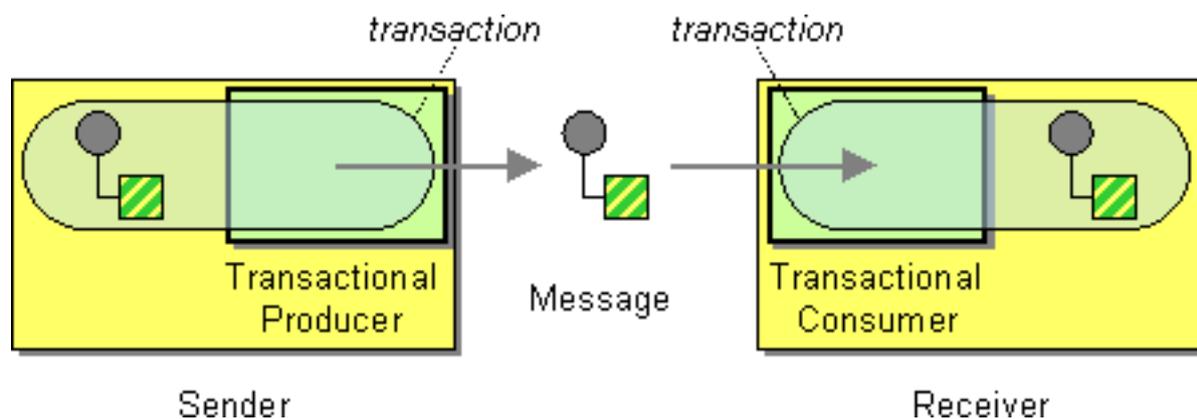
Transaction Oriented Endpoints (Camel Toes) like Section 3.25, "JMS" support using a transaction for both inbound and outbound message exchanges. Endpoints that support transactions will participate in the current transaction context that they are called from.

Configuration of Redelivery

The redelivery in transacted mode is **not** handled by Camel but by the backing system (the transaction manager). In such cases you should resort to the backing system how to configure the redelivery.

You should use the SpringRouteBuilder to setup the routes since you will need to setup the Spring context with the TransactionTemplate s that will define the transaction manager configuration and policies.

For inbound endpoint to be transacted, they normally need to be configured to use a Spring PlatformTransactionManager. In the case of the JMS component, this can be done by looking it up in the Spring context.

You first define needed object in the Spring configuration.

```
<bean id="jmsTransactionManager"
    class="org.springframework.jms.connection.JmsTransactionManager">
    <property name="connectionFactory" ref="jmsConnectionFactory" />
```

```
  </bean>

  <bean id="jmsConnectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
  </bean>
```

Then you look them up and use them to create the JmsComponent.

```
PlatformTransactionManager transactionManager =
    (PlatformTransactionManager) spring.getBean("jmsTransactionManager");
    ConnectionFactory connectionFactory = (ConnectionFactory) spring.getBean(
        "jmsConnectionFactory");
    JmsComponent component = JmsComponent.jmsComponentTransacted(
        connectionFactory, transactionManager);
    component.getConfiguration().setConcurrentConsumers(1);
    ctx.addComponent("activemq", component);
```

2.50.1.1. Transaction Policies

Outbound endpoints will automatically enlist in the current transaction context. But what if you do not want your outbound endpoint to enlist in the same transaction as your inbound endpoint? The solution is to add a Transaction Policy to the processing route. You first have to define transaction policies that you will be using. The policies use a Spring TransactionTemplate under the covers for declaring the transaction demarcation to use. So you will need to add something like the following to your Spring XML:

```
<bean id="PROPAGATION_REQUIRED"
class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="jmsTransactionManager"/>
  </bean>

  <bean id="PROPAGATION_REQUIRES_NEW"
    class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="jmsTransactionManager"/>
    <property name="propagationBehaviorName"
        value="PROPAGATION_REQUIRES_NEW"/>
  </bean>
```

Then in your SpringRouteBuilder, you just need to create new SpringTransactionPolicy objects for each of the templates.

```
public void configure() {
    ...
    Policy requried = bean(SpringTransactionPolicy.class,
        "PROPAGATION_REQUIRED"));
    Policy requirenew = bean(SpringTransactionPolicy.class,
        "PROPAGATION_REQUIRES_NEW"));
    ...
}
```

Once created, you can use the Policy objects in your processing routes:

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

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

2.50.1.2. OSGi Blueprint

If you are using OSGi Blueprint then you most likely have to explicit declare a policy and refer to the policy from the transacted in the route.

```
<bean id="required"
    class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="jmsTransactionManager"/>
    <property name="propagationBehaviorName" value="PROPAGATION_REQUIRED"/>
</bean>
```

And then refer to "required" from the route:

```
<route>
  <from uri="activemq:queue:foo"/>
  <transacted ref="required"/>
  <to uri="activemq:queue:bar"/>
</route>
}
```

2.50.2. Camel 2.0 - Database Sample

In this sample we want to ensure that two endpoints are under transaction control. These two endpoints insert data into a database. The sample appears in full in a unit test.

First of all we setup the normal Spring configuration file. Here we have defined a DataSource to the HSQLDB and a most importantly the Spring DataSource TransactionManager that is doing the heavy lifting of ensuring our transactional policies. You are of course free to use any of the Spring based TransactionMananger, eg. if you are in a full blown J2EE container you could use JTA or the WebLogic or WebSphere specific managers.

As we use the new convention over configuration we do **not** need to configure a transaction policy bean, so we do not have any `PROPAGATION_REQUIRED` beans. All the beans needed to be configured is **standard** Spring beans only, eg. there are no Camel specific configuration at all.

```
<!-- this example uses JDBC so we define a data source -->
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:mem:camel"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
</bean>

<!-- Spring transaction manager -->
<!-- this is the transaction manager Camel will use for transacted routes -->
<bean id="txManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<!-- bean for book business logic -->
<bean id="bookService"
    class="org.apache.camel.spring.interceptor.BookService">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

Then we are ready to define our Camel routes. We have two routes: 1 for success conditions, and 1 for a forced rollback condition. This is after all based on a unit test. Notice that we mark each route as transacted using the **transacted** tag.

```xml
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:okay"/>
        <!-- We mark this route as transacted. Camel will lookup the Spring
                transaction manager and use it by default. We can optimally
                pass in arguments to specify a policy to use that is configured
                with a Spring transaction manager of choice. However Camel
                supports convention over configuration as we can just use the
                defaults out of the box suitable for most situations -->
        <transacted/>
        <setBody>
            <constant>Tiger in Action</constant>
        </setBody>
        <bean ref="bookService"/>
        <setBody>
            <constant>Elephant in Action</constant>
        </setBody>
        <bean ref="bookService"/>
    </route>

    <route>
        <from uri="direct:fail"/>
        <!-- we mark this route as transacted. See comments above. -->
        <transacted/>
        <setBody>
            <constant>Tiger in Action</constant>
        </setBody>
        <bean ref="bookService"/>
        <setBody>
            <constant>Donkey in Action</constant>
        </setBody>
        <bean ref="bookService"/>
    </route>
</camelContext>
```

That is all that is needed to configure a Camel route as being transacted. Just remember to use the **transacted** DSL. The rest is standard Spring XML to setup the transaction manager.

2.50.3. Camel 2.0 - JMS Sample

In this sample we want to listen for messages on a queue and process the messages with our business logic Java code and send them along. Since it is based on a unit test the destination is a mock endpoint.

First we configure the standard Spring XML to declare a JMS connection factory, a JMS transaction manager and our ActiveMQ component that we use in our routing.

```xml
<!-- setup JMS connection factory -->
<bean id="jmsConnectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL"
        value="vm://localhost?broker.persistent=false&broker.useJmx=false"/>
</bean>
```

```xml
<!-- setup Spring jms TX manager -->
<bean id="jmsTransactionManager"
    class="org.springframework.jms.connection.JmsTransactionManager">
    <property name="connectionFactory" ref="jmsConnectionFactory"/>
</bean>


<!-- define our activemq component -->
<bean id="activemq"
    class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="connectionFactory" ref="jmsConnectionFactory"/>
    <!-- define the jms consumer/producer as transacted -->
    <property name="transacted" value="true"/>
    <!-- setup the transaction manager to use -->
    <!-- if not provided then Camel will automatically use a
         JmsTransactionManager, however if you for instance use a JTA
         transaction manager then you must configure it -->
    <property name="transactionManager" ref="jmsTransactionManager"/>
</bean>
```

And then we configure our routes. Notice that all we have to do is mark the route as transacted using the **transacted** tag.

```xml
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- disable JMX during testing -->
    <jmxAgent id="agent" disabled="true"/>
    <route>
        <!-- 1: from the jms queue -->
        <from uri="activemq:queue:okay"/>
        <!-- 2: mark this route as transacted -->
        <transacted/>
        <!-- 3: call our business logic that is myProcessor -->
        <process ref="myProcessor"/>
        <!-- 4: if success then send it to the mock -->
        <to uri="mock:result"/>
    </route>
</camelContext>

<bean id="myProcessor"
    class="org.apache.camel.component.jms.tx.JMSTransactionalClientTest \\
    $MyProcessor"/>
```

Transaction error handler

When a route is marked as transacted using **transacted** Camel will automatically use the TransactionErrorHandler as Error Handler. It supports basically the same feature set as the DefaultErrorHandler, so you can for instance use Exception Clause as well.

2.50.4. Using multiple routes with different propagation behaviors

Suppose you want to route a message through two routes and by which the second route should run in its own transaction. How do you do that? You use propagation behaviors for that where you configure it as follows:

- The first route use PROPAGATION_REQUIRED

- The second route use `PROPAGATION_REQUIRES_NEW`

This is configured in the Spring XML file:

```
<bean id="PROPAGATION_REQUIRED"
    class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager"/>
    <property name="propagationBehaviorName" value="PROPAGATION_REQUIRED"/>
</bean>

<bean id="PROPAGATION_REQUIRES_NEW"
    class="org.apache.camel.spring.spi.SpringTransactionPolicy">
    <property name="transactionManager" ref="txManager"/>
    <property name="propagationBehaviorName"
        value="PROPAGATION_REQUIRES_NEW"/>
</bean>
```

Then in the routes you use transacted DSL to indicate which of these two propagations it uses.

```
from("direct:mixed")
    // using required
    .transacted("PROPAGATION_REQUIRED")
    // all these steps will be okay
    .setBody(constant("Tiger in Action")).beanRef("bookService")
    .setBody(constant("Elephant in Action")).beanRef("bookService")
    // continue on route 2
    .to("direct:mixed2");

from("direct:mixed2")
    // tell Camel that if this route fails then only rollback this
    // last route by using (rollback only *last*)
    .onException(Exception.class).markRollbackOnlyLast().end()
    // using a different propagation which is requires new
    .transacted("PROPAGATION_REQUIRES_NEW")
    // this step will be okay
    .setBody(constant("Lion in Action")).beanRef("bookService")
    // this step will fail with donkey
    .setBody(constant("Donkey in Action")).beanRef("bookService");
```

Notice how we have configured the `onException` in the second route to indicate in case of any exceptions we should handle it and just rollback this transaction. This is done using the `markRollbackOnlyLast` which tells Camel to only do it for the current transaction and not globally.

2.51. Validate

Validate uses an expression or predicates to validate the contents of a message. It is useful for ensuring that messages are valid before attempting to process them.

You can use the validate DSL with all kind of Predicates and Expressions. Validate evaluates the Predicate/Expression and if it is false a `PredicateValidationException` is thrown. If it is true message processing continues.

2.51.1. Using from Java DSL

The route below will read the file contents and validate them against a regular expression.

```
from("file://inbox")
  .validate(body(String.class).regex("^\\w{10}\\,\\d{2}\\,\\w{24}$"))
.to("bean:MyServiceBean.processLine");
```

Validate is not limited to the message body. You can also validate the message header.

```
from("file://inbox")
  .validate(header("bar").isGreaterThan(100))
.to("bean:MyServiceBean.processLine");
```

You can also use validate together with simple.

```
from("file://inbox")
  .validate(simple("${in.header.bar} == 100"))
.to("bean:MyServiceBean.processLine");
```

2.51.2. Using from Spring DSL

To use validate in the Spring DSL, the easiest way is to use simple expressions.

```
<route>
  <from uri="file://inbox"/>
  <validate>
    <simple>${body} regex ^\\w{10}\\,\\d{2}\\,\\w{24}$</simple>
  </validate>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>

<bean id="myServiceBean" class="com.mycompany.MyServiceBean"/>
```

The XML DSL to validate the message header would looks like this:

```
<route>
  <from uri="file://inbox"/>
  <validate>
    <simple>${in.header.bar} == 100</simple>
  </validate>
  <beanRef ref="myServiceBean" method="processLine"/>
</route>

<bean id="myServiceBean" class="com.mycompany.MyServiceBean"/>
```

2.52. Wire Tap

2.52.1. Wire Tap

The Wire Tap from the EIP patterns allows you to route messages to a separate tap location while it is forwarded to the ultimate destination.
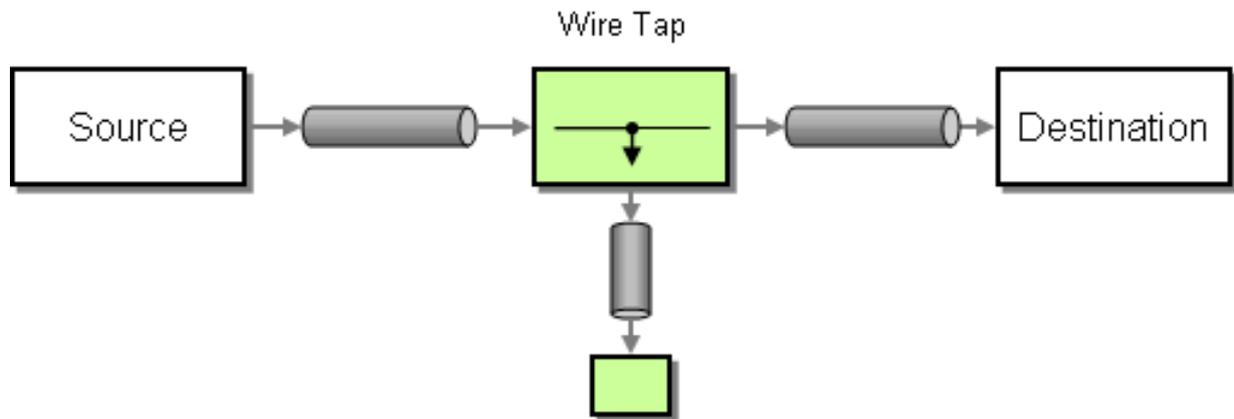
Wire Tap

Options:

Name	Default Value	Description
uri		The URI of the endpoint to which the wire-tapped message will be sent. You should use either uri or ref.
ref		Reference identifier of the endpoint to which the wire-tapped message will be sent. You should use either uri or ref.
executorServiceRef		Reference identifier of a custom Thread Pool to use when processing the wire-tapped messages. If not set, Camel will use a default thread pool.
processorRef		Reference identifier of a custom Processor to use for creating a new message (e.g., the "send a new message" mode).
copy	true	Whether to copy the Exchange before wire-tapping the message.
onPrepareRef		Reference identifier of a custom Processor to prepare the copy of the Exchange to be wire-tapped. This allows you to do any custom logic, such as deep-cloning the message payload.

2.52.2. WireTap node

Camel's WireTap node supports two flavors when tapping an Exchange.

• With the traditional Wire Tap, Camel will copy the original Exchange and set its Exchange Pattern to InOnly, as we want the tapped Exchange to be sent in a fire and forget style. The tapped Exchange is then sent in a separate thread so it can run in parallel with the original.

• Camel also provides an option of sending a new Exchange allowing you to populate it with new values. See the Camel Website for dynamically maintained examples of this pattern in use.

2.52.2.1. Sending a copy (traditional wire tap)

Using the Fluent Builders

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

Using the Spring XML Extensions

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

Chapter 3. Components

The following Camel components are discussed within this guide:

Component / ArtifactId / URI	Description
Section 3.1, "ActiveMQ" / activemq-camel `activemq:[topic:]destinationName`	For JMS Messaging with Apache ActiveMQ
Section 3.2, "Atom" / camel-atom `atom:uri`	Working with Apache Abdera for atom integration, such as consuming an atom feed.
Section 3.3, "Bean" / camel-core `bean:beanName[?method=someMethod]`	Uses the Camel Bean Binding to bind message exchanges to beans in the Camel Registry. Is also used for exposing and invoking POJO (Plain Old Java Objects).
Section 3.4, "Cache" / camel-cache `cache://cachename[?options]`	The cache component facilitates creation of caching endpoints and processors using EHCache as the cache implementation.
Section 3.5, "Class" / camel-core `class:className[?method=someMethod]`	Uses the Camel Bean Binding to bind message exchanges to beans in the Camel Registry. Is also used for exposing and invoking POJOs (Plain Old Java Objects).

Section 3.6, "Context" / camel-context
 	
`context:camelContextId:`
`localEndpointName`
	Used to refer to endpoints within a separate CamelContext to provide a simple black box composition approach so that routes can be combined into a CamelContext and then used as a black box component inside other routes in other CamelContexts

Section 3.7, "Crypto (Digital Signatures)"
 	
`crypto:sign:name[?options]`
`crypto:verify:name[?options]`
	Used to sign and verify exchanges using the Signature Service of the Java Cryptographic Extension.

Section 3.8, "CXF" / camel-cxf	Working with Apache CXF for web services integration

---

Component / ArtifactId / URI	Description
`cxf:address[?serviceClass=...]`	
Section 3.9, "CXF Bean Component" / camel-cxf `cxf:bean name`	Process the exchange using a JAX WS or JAX RS annotated bean from the registry. Requires less configuration than the above CXF Component
Section 3.10, "CXFRS" / camel-cxf `cxfrs:address[?resourcesClasses=...]`	Working with Apache CXF for REST services integration
Section 3.11, "Direct" / camel-core `direct:name`	Synchronous call to another endpoint

Section 3.12, "Event" / camel-spring
 	
`event://default`
`spring-event://default`
	Working with Spring ApplicationEvents

Section 3.13, "Exec" / camel-exec `exec://executable[?options]`	For executing system commands
Section 3.14, "File" / camel-core `file://nameOfFileOrDirectory`	Sending messages to a file or polling a file or directory.
Section 3.15, "Flatpack" / camel-flatpack `flatpack:[fixed|delim]:configFile`	Processing fixed width or delimited files or messages using the FlatPack library
Section 3.16, "Freemarker" / camel-freemarker `freemarker:someTemplateResource`	Generates a response using a Freemarker template
Section 3.17, "FTP" / camel-ftp `ftp://host[:port]/fileName`	Sending and receiving files over FTP.
Section 3.17, "FTP" / camel-ftp (FTPS) `ftps://host[:port]/fileName`	Sending and receiving files over FTP Secure (TLS and SSL).
Section 3.18, "Hibernate" / camel-hibernate in camel-extra `hibernate://entityName`	For using a database as a queue via the Hibernate library
Section 3.19, "Hl7" `mina:tcp://hostname[:port]`	For working with the HL7 MLLP protocol and the HL7 model using the HAPI library.
Section 3.20, "HTTP4" / camel-http4 `http4://hostname[:port]`	For calling out to external HTTP servers using Apache HTTP Client 4.x
Section 3.31, "Mail" / camel-mail `imap://hostname[:port]`	Receiving email using IMap
Section 3.21, "Jasypt" / camel-jasypt `jasypt: uri`	Simplified on-the-fly encryption library, integrated with Camel.

Section 3.22, "JCR" / camel-jcr
 	
`jcr://user:password@repository/`
`path/to/node`
	Storing a message in a JCR (JSR-170) compliant repository like Apache Jackrabbit

Component / ArtifactId / URI	Description
Section 3.23, "JDBC" / camel-jdbc `jdbc:dataSourceName?options`	For performing JDBC queries and operations
Section 3.24, "Jetty" / camel-jetty `jetty:url`	For exposing services over HTTP
Section 3.25, "JMS" / camel-jms `jms:[topic:]destinationName`	Working with JMS providers
Section 3.26, "JMX" / camel-jmx `jmx://platform?options`	For working with JMX notification listeners
Section 3.27, "JPA" / camel-jpa `jpa://entityName`	For using a database as a queue via the JPA specification for working with OpenJPA, Hibernate or TopLink
Section 3.28, "Jsch" / camel-jsch `scp://localhost/destination`	Support for the scp protocol.
Section 3.29, "Log" / camel-core `log:loggingCategory[?level=ERROR]`	Uses Jakarta Commons Logging to log the message exchange to some underlying logging system like log4j
Section 3.30, "Lucene" / camel-lucene `lucene:searcherName:insert [?analyzer=<analyzer>]`	Uses Apache Lucene to perform Java-based indexing and full text based searches using advanced analysis/tokenization capabilities
Section 3.31, "Mail" / camel-mail `mail://user-info@host:port`	Sending and receiving email
Section 3.32, "Mock" / camel-core `mock:name`	For testing routes and mediation rules using mocks
Section 3.31, "Mail" / camel-mail `pop3://user-info@host:port`	Receiving email using POP3 and JavaMail
Section 3.33, "MyBatis" / camel-mybatis `mybatis://statementName`	Performs a query, poll, insert, update or delete in a relational database using MyBatis
Section 3.34, "Properties" / camel-core `properties://key[?options]`	The properties component facilitates using property placeholders directly in endpoint uri definitions.
Section 3.35, "Quartz" / camel-quartz `quartz://groupName/timerName`	Provides a scheduled delivery of messages using the Quartz scheduler
Section 3.36, "Ref" / camel-core `ref:name`	Component for lookup of existing endpoints bound in the Camel Registry.
Section 3.37, "RMI" / camel-rmi `rmi://host[:port]`	Working with RMI
Section 3.38, "RSS" / camel-rss `rss:uri`	Working with ROME for RSS integration, such as consuming an RSS feed.

Component / ArtifactId / URI	Description
Section 3.39, "SEDA" / camel-core `seda:name`	Asynchronous call to another endpoint in the same Camel Context
Section 3.40, "Servlet" / camel-servlet `servlet:uri`	For exposing services over HTTP through the servlet which is deployed into the Web container.
Section 3.17, "FTP" / camel-ftp (SFTP) `sftp://host[:port]/fileName`	Sending and receiving files over SFTP (FTP over SSH).
Section 3.31, "Mail" / camel-mail `smtp://user-info@host[:port]`	Sending email using SMTP and JavaMail
Section 3.42, "SMPP" / camel-smpp `smpp://user-info@host[:port]?options`	To send and receive SMS using Short Messaging Service Center using the JSMPP library
Section 3.43, "SNMP" / camel-snmp `snmp://host[:port]?options`	Polling OID values and receiving traps using SNMP via SNMP4J library
Section 3.44, "Spring Integration" / camel-spring-integration `spring-integration:defaultChannelName`	The bridge component of Camel and Spring Integration
Section 3.46, "SQL Component" / camel-sql `sql:select * from table where id=#`	Performing SQL queries using JDBC
Section 3.47, "SSH" / camel-ssh `ssh:[username[:password]@]host[:port][?options]`	For sending commands to a SSH server
Section 3.48, "Stub" `stub:someOtherCamelUri`	Allows you to stub out some physical middleware endpoint for easier testing or debugging
Section 3.49, "Test" / camel-spring `test:expectedMessagesEndpointUri`	Creates a Section 3.32, "Mock" endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint
Section 3.50, "Timer" / camel-core `timer://name`	A timer endpoint
Section 3.51, "Velocity" / camel-velocity `velocity:someTemplateResource`	Generates a response using an Apache Velocity template
Section 3.52, "VM" / camel-core `vm:name`	Asynchronous call to another endpoint in the same JVM
Section 3.53, "XQuery Endpoint" / camel-saxon `xquery:someXQueryResource`	Generates a response using an XQuery template
Section 3.54, "XSLT" / camel-spring `xslt:someTemplateResource`	Generates a response using an XSLT template

3.1. ActiveMQ

The ActiveMQ component allows messages to be sent to a JMS Queue or Topic or messages to be consumed from a JMS Queue or Topic using Apache ActiveMQ.

This component is based on JMS Component and uses Spring's JMS support for declarative transactions, using Spring's `JmsTemplate` for sending and a `MessageListenerContainer` for consuming. All the options from the Section 3.25, "JMS" component also apply for this component.

To use this component make sure you have the `activemq.jar` or `activemq-core.jar` on your classpath along with any Camel dependencies such as `camel-core.jar`, `camel-spring.jar` and `camel-jms.jar`.

3.1.1. URI format

```
activemq:[queue:|topic:]destinationName
```

where **destinationName** is an ActiveMQ queue or topic name. By default, the **destinationName** is interpreted as a queue name. For example, to connect to the queue, `FOO.BAR`, use:

```
activemq:FOO.BAR
```

You can include the optional `queue:` prefix, if you prefer:

```
activemq:queue:FOO.BAR
```

To connect to a topic, you must include the `topic:` prefix. For example, to connect to the topic, `Stocks.Prices`, use:

```
activemq:topic:Stocks.Prices
```

3.1.2. Options

See Options on the Section 3.25, "JMS" component as all these options also apply for this component.

3.1.3. Configuring the Connection Factory

This test case shows how to add an ActiveMQComponent to the CamelContext using the activeMQComponent() method while specifying the brokerURL used to connect to ActiveMQ.

```
camelContext.addComponent("activemq", activeMQComponent(
      "vm://localhost?broker.persistent=false"));
```

3.1.4. Configuring the Connection Factory using Spring XML

You can configure the ActiveMQ broker URL on the ActiveMQComponent as follows

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

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

3.1.5. Using connection pooling

When sending to an ActiveMQ broker using Camel it is recommended to use a pooled connection factory to efficiently handle pooling of JMS connections, sessions and producers. This is documented on the ActiveMQ Spring Support page.

You can grab ActiveMQ's `org.apache.activemq.pool.PooledConnectionFactory` with Maven:

```
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-pool</artifactId>
    <version>5.3.2</version>
</dependency>
```

And then setup the **activemq** Camel component as follows:

```
<bean id="jmsConnectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<bean id="pooledConnectionFactory"
    class="org.apache.activemq.pool.PooledConnectionFactory">
    <property name="maxConnections" value="8" />
    <property name="maximumActive" value="500" />
    <property name="connectionFactory" ref="jmsConnectionFactory" />
</bean>

<bean id="jmsConfig"
    class="org.apache.camel.component.jms.JmsConfiguration">
    <property name="connectionFactory" ref="pooledConnectionFactory"/>
    <property name="transacted" value="false"/>
    <property name="concurrentConsumers" value="10"/>
</bean>

<bean id="activemq"
    class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="configuration" ref="jmsConfig"/>
</bean>
```

3.1.6. Invoking MessageListener POJOs in a Camel route

The ActiveMQ component also provides a helper TypeConverter from a JMS MessageListener to a Processor. This means that the Bean component is capable of invoking any JMS MessageListener bean directly inside any route.

So for example you can create a MessageListener in JMS like this:

```
public class MyListener implements MessageListener {
   public void onMessage(Message jmsMessage) {
       // ...
   }
}
```

Then use it in your Camel route as follows

```
from("file://foo/bar").
  bean(MyListener.class);
```

That is, you can reuse any of the Camel Components and easily integrate them into your JMS MessageListener POJO.

3.1.7. Consuming Advisory Messages

ActiveMQ can generate Advisory messages which are put in topics that you can consume. Such messages can help you send alerts in case you detect slow consumers or to build statistics (number of messages/produced per day, etc.) The following Spring DSL example shows you how to read messages from a topic.

The below route starts by reading the topic *ActiveMQ.Advisory.Connection*. To watch another topic, simply change the name according to the name provided in ActiveMQ Advisory Messages documentation. The parameter `mapJmsMessage=false` allows for converting the `org.apache.activemq.command.ActiveMqMessage` object from the JMS queue. Next, the body received is converted into a String for the purposes of this example and a carriage return is added. Finally, the string is added to a file:

```
<route>
    <from uri="activemq:topic:ActiveMQ.Advisory.Connection?
mapJmsMessage=false" />

    <convertBodyTo type="java.lang.String"/>
    <transform>
        <simple>${in.body}&#13;</simple>
    </transform>
    <to uri="file://data/activemq/?fileExist=Append&amp;
fileName=advisoryConnection-${date:now:yyyyMMdd}.txt" />

</route>
```

If you consume a message on a queue, you should see the following files under the data/activemq folder :

advisoryConnection-20100312.txt advisoryProducer-20100312.txt

and containing string:

```
ActiveMQMessage {commandId = 0, responseRequired = false,
messageId = ID:dell-charles-3258-1268399815140-1:0:0:0:221,
originalDestination = null, originalTransactionId = null, producerId = ID:
dell-charles-3258-1268399815140-1:0:0:0, destination =
```

```
topic://ActiveMQ.Advisory.Connection, transactionId = null, expiration = 0,
timestamp = 0, arrival = 0, brokerInTime = 1268403383468, brokerOutTime =
1268403383468, correlationId = null, replyTo = null, persistent = false,
type = Advisory, priority = 0, groupID = null, groupSequence = 0,
targetConsumerId = null, compressed = false, userID = null, content = null,
marshalledProperties = org.apache.activemq.util.ByteSequence@17e2705,
dataStructure = ConnectionInfo {commandId = 1, responseRequired = true,
connectionId = ID:dell-charles-3258-1268399815140-2:50, clientId =
ID:dell-charles-3258-1268399815140-14:0, userName = , password = *****,
brokerPath = null, brokerMasterConnector = false, manageable = true,
clientMaster = true}, redeliveryCounter = 0, size = 0, properties =
{originBrokerName=master,
originBrokerId=ID:dell-charles-3258-1268399815140-0:0,
originBrokerURL=vm://master}, readOnlyProperties = true,
readOnlyBody = true, droppable = false}
```

3.1.8. Getting Component JAR

You will need these dependencies

- `camel-jms`

- `activemq-camel`

3.1.8.1. camel-jms

You **must** have the `camel-jms` as dependency as Section 3.1, "ActiveMQ" is an extension to the Section 3.25, "JMS" component.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jms</artifactId>
  <version>1.6.0</version>
</dependency>
```

The ActiveMQ Camel component is released with the ActiveMQ project itself. For Maven 2 users you simply need to add the following dependency to your project.

3.1.8.2. ActiveMQ 5.2 or later

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-camel</artifactId>
  <version>5.2.0</version>
</dependency>
```

3.1.8.3. ActiveMQ 5.1.0

For 5.1.0 it is in the activemq-core library

```
<dependency>
  <groupId>org.apache.activemq</groupId>
```

```
    <artifactId>activemq-core</artifactId>
    <version>5.1.0</version>
</dependency>
```

Alternatively you can download the component jar directly from the Maven repository:

- activemq-camel-5.2.0.jar

- activemq-core-5.1.0.jar

3.1.8.4. ActiveMQ 4.x

For this version you must use the Section 3.25, "JMS" component instead. Please be careful to use a pooling connection factory as described in the JmsTemplate Gotchas.

3.2. Atom

The **atom:** component is used for polling Atom feeds.

Camel will poll the feed every 60 seconds by default. **Note:** The component currently only supports polling (consuming) feeds.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-atom</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.2.1. URI format

`atom://atomUri[?options]`

where **atomUri** is the URI to the Atom feed to poll.

3.2.2. Options

Property	Default	Description
splitEntries	true	If `true` Camel will poll the feed and for the subsequent polls return each entry poll by poll. For example, if the feed contains seven entries then Camel will return the first entry on the first poll, the second entry on the next poll, until no more entries where as Camel will do a new update on the feed. If `false` then Camel will poll a fresh feed on every invocation.
filter	true	is only used by the split entries to filter the entries to return. Camel will default use the `UpdateDateFilter` that only returns new entries from the feed. So the client consuming from the feed

Property	Default	Description
		never receives the same entry more than once. The filter will return the entries ordered by the newest last.
lastUpdate	null	Is only used when filter=true. It defines the starting timestamp for selecting newer entries (uses the entry.updated timestamp). Syntax format is: yyyy-MM-ddTHH:MM:ss. Example: 2007-12-24T17:45:59.
throttleEntries	true	Sets whether all entries identified in a single feed poll should be delivered immediately. If true, only one entry is processed per consumer.delay. Only applicable when splitEntries is set to true.
feedHeader	true	Sets whether to add the Abdera Feed object as a header.
sortEntries	false	If splitEntries is true, this sets whether to sort those entries by updated date.
consumer.delay	60000	Delay in milliseconds between each poll.
consumer.initialDelay	1000	Millis before polling starts.
consumer.userFixedDelay	false	If true, use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.

You can append query options to the URI in the following format, ?option=value&option=value&...

3.2.3. Exchange data format

Camel will set the In body on the returned Exchange with the entries. Depending on the splitEntries flag Camel will either return one Entry or a List<Entry>.

Option	Value	Behavior
splitEntries	true	Only a single entry from the currently being processed feed is set: exchange.in.body(Entry)
splitEntries	false	The entire list of entries from the feed is set: exchange.in.body(List<Entry>)

Camel can set the Feed object on the In header (see feedHeader option to disable this).

3.2.4. Message Headers

Camel atom uses these headers:

Header	Description
CamelAtomFeed	When consuming the org.apache.abdera.model.Feed object is set to this header.

3.2.5. Samples

In this sample we poll Joe Bloggs' blog.

```
from("atom://http://joebloggs.blogspot.com/feeds/posts/default")
                .to("seda:feeds");
```

In this sample we want to filter only good blogs we like to a SEDA queue. The sample also shows how to setup
Camel standalone, not running in any Container or using Spring.

```
// This is the CamelContext that is the heart of Camel
private CamelContext context;

protected CamelContext createCamelContext() throws Exception {

    // First we register a blog service in our bean registry
    SimpleRegistry registry = new SimpleRegistry();
    registry.put("blogService", new BlogService());

    // Then we create the Camel context with our bean registry
    context = new DefaultCamelContext(registry);

    // Then we add all the routes we need using the
    // route builder DSL syntax
    context.addRoutes(createMyRoutes());

    return context;
}

/**
 * This is the route builder where we create our routes using the Camel DSL
 */
protected RouteBuilder createMyRoutes() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // We pool the atom feeds from the source for further processing
            // in the seda queue. We set the delay to 1 second for each pool
            // as this is a unit test also and we cannot wait the default poll
            // interval of 60 seconds.
            // Using splitEntries=true will (during polling) only fetch one
            // Atom Entry at any given time.
            // As the feed.atom file contains seven entries, using this will
            // require seven polls to fetch the entire content. When Camel has
            // reached the end of entries it will refresh the atom feed from
            // URI source and restart - but as Camel by default uses the
            // UpdatedDateFilter it will only deliver new blog entries to
            // "seda:feeds". So only when Joe Bloggs updates his blog with
            // a new entry, Camel will create an exchange for the seda:feeds.
            from("atom:file:src/test/data/feed.atom?splitEntries=
                true&consumer.delay=1000").to("seda:feeds");

            // From the feeds we filter each blog entry by using our blog
            // service class
            from("seda:feeds").filter().method("blogService", "isGoodBlog")
                .to("seda:goodBlogs");
            // And the good blogs is moved to a mock queue as this sample is
            // also used for unit testing. This is one of the strengths in
            // Camel- that you can also use the mock endpoint for your unit
            // tests.
            from("seda:goodBlogs").to("mock:result");
        }
    };
```

```
}

/**
 * This is the  junit test method that does the assertion that our
 * routes is working as expected
 */
@Test
public void testFiltering() throws Exception {
    // create and start Camel
    context = createCamelContext();
    context.start();

    // Get the mock endpoint
    MockEndpoint mock = context.getEndpoint("mock:result",
        MockEndpoint.class);

    // There should be at least two good blog entries from the feed
    mock.expectedMinimumMessageCount(2);
    // Asserts that the above expectations is true, will throw assertions
    // exception if it failed. Camel will by default wait max 20 seconds
    // for the assertions to be true. If the conditions are true sooner,
    // Camel will continue.
    mock.assertIsSatisfied();

    // stop Camel after use
    context.stop();
}

/**
 * Services for blogs
 */
public class BlogService {

    /**
     * Tests the blogs to see if it is a good blog entry or not
     */
    public boolean isGoodBlog(Exchange exchange) {
        Entry entry = exchange.getIn().getBody(Entry.class);
        String title = entry.getTitle();

        // We like blogs about Camel
        boolean good = title.toLowerCase().contains("camel");
        return good;
    }

}
```

3.3. Bean

The **bean:** component binds beans to Camel message exchanges.

3.3.1. URI format

```
bean:beanID[?options]
```

where **beanID** can be any string which is used to look up the bean in the Camel Registry.

3.3.2. Options

Name	Type	Default	Description
method	String	null	The method name from the bean that will be invoked. If not provided, Camel will try to pick the method itself. In case of ambiguity an exception will be thrown. See Camel Bean Binding for more details.
cache	boolean	false	If enabled, Camel will cache the result of the first Registry look-up. Cache can be enabled if the bean in the Registry is defined as a singleton scope.
multi-Parameter-Array	boolean	false	How to treat the parameters which are passed from the message body; if it is true, the In message body should be an array of parameters.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.3.3. Using

The object instance that is used to consume messages must be explicitly registered with the Camel Registry. For example, if you are using Spring you must define the bean in the Spring configuration, `spring.xml`; or if you don't use Spring, by registering the bean in JNDI, as described here:

```
// let's populate the context with the services we need
// note that we could just use a spring.xml file to avoid this step
JndiContext context = new JndiContext();
context.bind("bye", new SayService("Good Bye!"));

CamelContext camelContext = new DefaultCamelContext(context);
```

Once an endpoint has been registered, you can build Camel routes that use it to process exchanges.

```
// let's add simple route
camelContext.addRoutes(new RouteBuilder() {
    public void configure() {
        from("direct:hello").to("bean:bye");
    }
});
```

Note: A **bean:** endpoint cannot be defined as the input to the route; that is you cannot consume from it, you can only route from some inbound message endpoint to the bean endpoint as output. So consider using a **direct:** or **queue:** endpoint as the input.

You can use the `createProxy()` methods on ProxyHelper to create a proxy that will generate BeanExchanges and send them to any endpoint:

```
Endpoint endpoint = camelContext.getEndpoint("direct:hello");
ISay proxy = ProxyHelper.createProxy(endpoint, ISay.class);
String rc = proxy.say();
assertEquals("Good Bye!", rc);
```

And the same route using Spring DSL:

```
<route>
   <from uri="direct:hello">
   <to uri="bean:bye"/>
</route>
```

3.3.4. Bean as endpoint

Camel also supports invoking Section 3.3, "Bean" as an Endpoint. In the route below:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="myBean"/>
    <to uri="mock:results"/>
  </route>
</camelContext>

<bean id="myBean" class="org.apache.camel.spring.bind.ExampleBean"/>
```

What happens is that when the exchange is routed to the `myBean` Camel will use the Bean Binding to invoke the bean. The source for the bean is just a plain POJO:

```
public class ExampleBean {

    public String sayHello(String name) {
        return "Hello " + name + "!";
    }
}
```

Camel will use the Bean Binding to invoke the `sayHello` method, by converting the Exchange's In body to the `String` type and storing the output of the method on the Exchange Out body.

3.3.5. Bean Binding

How bean methods to be invoked are chosen (if they are not specified explicitly through the **method** parameter) and how parameter values are constructed from the Message are all defined by the Bean Binding mechanism. This is used throughout all of the various Bean Integration mechanisms in Camel.

3.4. Cache

The **cache** component enables you to perform caching operations using EHCache as the Cache Implementation. The cache itself is created on demand or if a cache of that name already exists then it is simply utilized with its original settings.

This component supports producer and event based consumer endpoints.

The Cache consumer is an event based consumer and can be used to listen and respond to specific cache activities. If you need to perform selections from a pre-existing cache, use the processors defined for the cache component.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-cache</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.4.1. URI format

`cache://cacheName[?options]`

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.4.2. Options

Name	Default Value	Description
`maxElementsInMemory`	`1000`	The numer of elements that may be stored in the defined cache
`memoryStore-` `EvictionPolicy`	`MemoryStore-` `EvictionPolicy`	The number of elements that may be stored in the defined cache. Options include • MemoryStoreEvictionPolicy.LFU - Least frequently used • MemoryStoreEvictionPolicy.LRU - Least recently used • MemoryStoreEvictionPolicy.FIFO - first in first out, the oldest element by creation time
`overflowToDisk`	`true`	Specifies whether cache may overflow to disk
`eternal`	`false`	Sets whether elements are eternal. If eternal, timeouts are ignored and the element never expires.
`timeToLiveSeconds`	`300`	The maximum time between creation time and when an element expires. Is used only if the element is not eternal.
`timeToIdleSeconds`	`300`	The maximum amount of time between accesses before an element expires
`diskPersistent`	`false`	Whether the disk store persists between restarts of the Virtual Machine.
`diskExpiryThread-` `IntervalSeconds`	`120`	The number of seconds between runs of the disk expiry thread.
`cacheManagerFactory`	`null`	If you want to use a custom factory which instantiates and creates the EHCache `net.sf.ehcache.CacheManager`. Use type of abstract `org.apache.camel.component.cache. CacheManagerFactory`.

Name	Default Value	Description
`eventListenerRegistry`	`null`	Sets a list of EHCache net.sf.ehcache.event.CacheEventListener for all new caches - no need to define it per cache in EHCache xml config anymore. Use type of `org.apache.camel. component.cache. CacheEventListenerRegistry`.
`cacheLoaderRegistry`	`null`	Sets a list of org.apache.camel. component.cache. CacheLoaderWrapper that extends EHCache net.sf.ehcache.loader.CacheLoader for all new caches - no need to define it per cache in EHCache xml config anymore. Use type of `org.apache.camel. component.cache. CacheLoaderRegistry`
`key`	`null`	To configure using a cache key by default. If a key is provided in the message header, then the key from the header takes precedence.
`operation`	`null`	To configure using an cache operation by default. If an operation in the message header, then the operation from the header takes precedence.

3.4.3. Sending/Receiving Messages to/from the cache

3.4.3.1. Message Headers

Header	Description
`CamelCacheOperation`	The operation to be performed on the cache. These headers are removed from the exchange after the cache operation is performed. Valid options are • CamelCacheGet • CamelCacheCheck • CamelCacheAdd • CamelCacheUpdate • CamelCacheDelete • CamelCacheDeleteAll
`CamelCacheKey`	The cache key used to store the Message in the cache. The cache key is optional if the CamelCacheOperation is CamelCacheDeleteAll.

3.4.3.2. Cache Producer

Sending data to the cache involves the ability to direct payloads in exchanges to be stored in a pre-existing or created-on-demand cache. The mechanics of doing this involve

• setting the Message Exchange Headers shown above.

• ensuring that the Message Exchange Body contains the message directed to the cache

3.4.3.3. Cache Consumer

Receiving data from the cache involves the ability of the CacheConsumer to listen on a pre-existing or created-on-demand Cache using an event Listener and receive automatic notifications when any cache activity take place (i.e., Add, Update, Delete, or DeleteAll). Upon such an activity taking place

* an exchange containing Message Exchange Headers and a Message Exchange Body containing the just added/updated payload is placed and sent.

* in case of a CamelCacheDeleteAll operation, the Message Exchange Header CamelCacheKey and the Message Exchange Body are not populated.

3.4.3.4. Cache Processors

There are a set of nice processors with the ability to perform cache lookups and selectively replace payload content at the

* body

* token

* xpath level

3.4.4. Cache Usage Samples

3.4.4.1. Example 1: Configuring the cache

```
from("cache://MyApplicationCache" +
        "?maxElementsInMemory=1000" +
        "&memoryStoreEvictionPolicy=" +
        "MemoryStoreEvictionPolicy.LFU" +
        "&overflowToDisk=true" +
        "&eternal=true" +
        "&timeToLiveSeconds=300" +
        "&timeToIdleSeconds=true" +
        "&diskPersistent=true" +
        "&diskExpiryThreadIntervalSeconds=300")
```

3.4.4.2. Example 2: Adding keys to the cache

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("direct:start")
            .setHeader(CacheConstants.CACHE_OPERATION,
                constant(CacheConstants.CACHE_OPERATION_ADD))
            .setHeader(CacheConstants.CACHE_KEY,
                constant("Ralph_Waldo_Emerson"))
            .to("cache://TestCache1")
    }
};
```

3.4.4.3. Example 2: Updating existing keys in a cache

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
      from("direct:start")
         .setHeader(CacheConstants.CACHE_OPERATION,
            constant(CacheConstants.CACHE_OPERATION_UPDATE))
         .setHeader(CacheConstants.CACHE_KEY,
            constant("Ralph_Waldo_Emerson"))
         .to("cache://TestCache1")
   }
};
```

3.4.4.4. Example 3: Deleting existing keys in a cache

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
      from("direct:start")
         .setHeader(CacheConstants.CACHE_OPERATION,
            constant(CacheConstants.CACHE_DELETE))
         .setHeader(CacheConstants.CACHE_KEY,
            constant("Ralph_Waldo_Emerson"))
         .to("cache://TestCache1")
   }
};
```

3.4.4.5. Example 4: Deleting all existing keys in a cache

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
      from("direct:start")
         .setHeader(CacheConstants.CACHE_OPERATION,
            constant(CacheConstants.CACHE_DELETEALL))
         .to("cache://TestCache1");
   }
};
```

3.4.4.6. Example 5: Notifying any changes registering in a Cache to Processors and other Producers

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
      from("cache://TestCache1").process(new Processor() {
         public void process(Exchange exchange) throws Exception {
            String operation =
               (String) exchange.getIn().getHeader(
               CacheConstants.CACHE_OPERATION);
            String key = (String)
               exchange.getIn().getHeader(CacheConstants.CACHE_KEY);
            Object body = exchange.getIn().getBody();
            // Do something
```

```
        }
    })
  }
};
```

3.4.4.7. Example 6: Using Processors to selectively replace payload with cache values

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
      //Message Body Replacer
       from("cache://TestCache1")
          .filter(header(CacheConstants.CACHE_KEY).isEqualTo("greeting"))
          .process(new CacheBasedMessageBodyReplacer(
             "cache://TestCache1","farewell"))
          .to("direct:next");
       //Message Token replacer
       from("cache://TestCache1")
          .filter(header(CacheConstants.CACHE_KEY).isEqualTo("quote"))
          .process(new CacheBasedTokenReplacer(
            "cache://TestCache1","novel","#novel#"))
          .process(new CacheBasedTokenReplacer(
            "cache://TestCache1","author","#author#"))
          .process(new CacheBasedTokenReplacer(
            "cache://TestCache1","number","#number#"))
          .to("direct:next");

       //Message XPath replacer
       from("cache://TestCache1")
          .filter(header(CacheConstants.CACHE_KEY).isEqualTo("XML_FRAGMENT"))
          .process(new CacheBasedXPathReplacer(
            "cache://TestCache1","book1","/books/book1"))
          .process (new CacheBasedXPathReplacer(
            "cache://TestCache1","book2","/books/book2"))
          .to("direct:next");
   }
};
```

3.4.4.8. Example 7: Getting an entry from the Cache

```
from("direct:start")
   // Prepare headers
   .setHeader(CacheConstants.CACHE_OPERATION, constant(
          CacheConstants.CACHE_OPERATION_GET))
   .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson")).
   .to("cache://TestCache1").
    // Check if entry was not found
    .choice().when(header(
       CacheConstants.CACHE_ELEMENT_WAS_FOUND).isNull()).
       // If not found, get the payload and put it to cache
       .to("cxf:bean:someHeavyweightOperation")
       .setHeader(CacheConstants.CACHE_OPERATION, constant(
```

```
        CacheConstants.CACHE_OPERATION_ADD))
      .setHeader(CacheConstants.CACHE_KEY,
          constant("Ralph_Waldo_Emerson"))
      .to("cache://TestCache1")
   .end()
   .to("direct:nextPhase");
```

3.4.4.9. Example 8: Checking for an entry in the Cache

Note: The CHECK command tests existence of an entry in the cache but doesn't place a message in the body.

```
from("direct:start")
   // Prepare headers
   .setHeader(CacheConstants.CACHE_OPERATION,
          constant(CacheConstants.CACHE_OPERATION_CHECK))
   .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson")).
   .to("cache://TestCache1").
   // Check if entry was not found
   .choice().when(header(CacheConstants.CACHE_ELEMENT_WAS_FOUND).isNull()).
      // If not found, get the payload and put it to cache
      .to("cxf:bean:someHeavyweightOperation").
      .setHeader(CacheConstants.CACHE_OPERATION,
          constant(CacheConstants.CACHE_OPERATION_ADD))
      .setHeader(CacheConstants.CACHE_KEY, constant("Ralph_Waldo_Emerson"))
      .to("cache://TestCache1")
   .end();
```

3.4.5. Management of EHCache

EHCache has its own statistics and management from JMX .

Here's a snippet on how to expose them via JMX in a Spring application context:

```
<bean id="ehCacheManagementService"
   class="net.sf.ehcache.management.ManagementService"
   init-method="init" lazy-init="false">
   <constructor-arg>
      <bean class="net.sf.ehcache.CacheManager"
         factory-method="getInstance"/>
   </constructor-arg>
   <constructor-arg>
      <bean class="org.springframework.jmx.support.JmxUtils"
          factory-method="locateMBeanServer"/>
   </constructor-arg>
   <constructor-arg value="true"/>
   <constructor-arg value="true"/>
   <constructor-arg value="true"/>
   <constructor-arg value="true"/>
</bean>
```

Of course the same thing can be done in straight Java:

```
ManagementService.registerMBeans(
  CacheManager.getInstance(), mbeanServer, true, true, true, true);
```

You can get cache hits, misses, in-memory hits, disk hits, size stats this way. You can also change CacheConfiguration parameters on the fly.

3.5. Class

3.5.1. Class Component

The **class:** component binds beans to Camel message exchanges. It works in the same way as the Section 3.3, "Bean" component but instead of looking up beans from a Registry it creates the bean based on the class name.

3.5.1.1. URI format

```
class:className[?options]
```

where **className** is the fully qualified class name to create and use as bean.

3.5.1.2. Options

Name	Type	Default	Description
method	String	null	The method name that bean will be invoked. If not provided, Camel will try to pick the method itself. In case of ambiguity an exception is thrown. See Bean Binding for more details.
multi-Parameter-Array	boolean	false	How to treat the parameters which are passed from the message body; if it is true, the In message body should be an array of parameters.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.5.1.3. Using

You simply use the **class** component just as the Section 3.3, "Bean" component but by specifying the fully qualified classname instead. For example to use the MyFooBean you have to do as follows:

```
from("direct:start")
    .to("class:org.apache.camel.component.bean.MyFooBean")
    .to("mock:result");
```

You can also specify which method to invoke on the MyFooBean, for example hello :

```
from("direct:start")
    .to("class:org.apache.camel.component.bean.MyFooBean?method=hello")
    .to("mock:result");
```

3.5.2. Setting properties on the created instance

In the endpoint uri you can specify properties to set on the created instance, for example, if it has a `setPrefix` method:

```
from("direct:start")
   .to("class:org.apache.camel.component.bean.MyPrefixBean?prefix=Bye")
   .to("mock:result");
```

You can also use the # syntax to refer to properties to be looked up in the Registry .

```
from("direct:start")
   .to("class:org.apache.camel.component.bean.MyPrefixBean?cool=#foo")
   .to("mock:result");
```

This will lookup a bean from the Registry with the id `foo` and invoke the `setCool` method on the created instance of the `MyPrefixBean` class.

See more

See more details at the Section 3.3, "Bean" component as the **class** component works in much the same way.

3.6. Context

Available as of Camel 2.7

The **context** component allows you to create new Camel Components from a CamelContext with a number of routes which is then treated as a black box, allowing you to refer to the local endpoints within the component from other CamelContexts.

It is similar to the Routebox component in idea, though the Context component tries to be really simple for end users; just a simple convention over configuration approach to refer to local endpoints inside the CamelContext Component.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-context</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.6.1. URI format

```
context:camelContextId:localEndpointName[?options]
```

Or you can omit the "context:" prefix.

```
camelContextId:localEndpointName[?options]
```

• **camelContextId** is the ID you used to register the CamelContext into the Registry.

- **localEndpointName** can be a valid Camel URI evaluated within the black box CamelContext. Or it can be a logical name which is mapped to any local endpoints. For example if you locally have endpoints like **direct:invoices** and **seda:purchaseOrders** inside a CamelContext of id **supplyChain**, then you can just use the URIs **supplyChain:invoices** or **supplyChain:purchaseOrders** to omit the physical endpoint kind and use pure logical URIs.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.6.2. Example

In this example we'll create a black box context, then we'll use it from another CamelContext.

3.6.2.1. Defining the context component

First you need to create a CamelContext, add some routes in it, start it and then register the CamelContext into the Registry (JNDI, Spring, Guice or OSGi etc).

This can be done in the usual Camel way from this test case (see the createRegistry() method); this example shows Java and JNDI being used:

```
// let's create our black box as a Camel context and a set of routes
DefaultCamelContext blackBox = new DefaultCamelContext(registry);
blackBox.setName("blackBox");
blackBox.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        // receive purchase orders, let's process it in some way then send
        // an invoice to our invoice endpoint
        from("direct:purchaseOrder")
            .setHeader("received")
            .constant("true")
            .to("direct:invoice");
    }
});
blackBox.start();

registry.bind("accounts", blackBox);
```

Notice in the above route we are using pure local endpoints (**direct** and **seda**). Also note we expose this CamelContext using the **accounts** ID. We can do the same thing in Spring via:

```
<camelContext id="accounts" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:purchaseOrder"/>
        ...
        <to uri="direct:invoice"/>
    </route>
</camelContext>
```

3.6.2.2. Using the context component

Then in another CamelContext we can then refer to this "accounts black box" by just sending to **accounts:purchaseOrder** and consuming from **accounts:invoice** .

If you prefer to be more verbose and explicit you could use **context:accounts:purchaseOrder** or even **context:accounts:direct://purchaseOrder** if you prefer. But using logical endpoint URIs is preferred as it hides the implementation detail and provides a simple logical naming scheme.

For example, if we wish to subsequently expose this accounts black box on some middleware (outside of the black box) we can do things like:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <!-- consume from an ActiveMQ into the black box -->
        <from uri="activemq:Accounts.PurchaseOrders"/>
        <to uri="accounts:purchaseOrders"/>
    </route>
    <route>
        <!-- let's send invoices from the black box -->
        <!-- to a different ActiveMQ Queue -->
        <from uri="accounts:invoice"/>
        <to uri="activemq:UK.Accounts.Invoices"/>
    </route>
</camelContext>
```

3.6.2.3. Naming endpoints

A context component instance can have many public input and output endpoints that can be accessed from outside its CamelContext. When there are many it is recommended that you use logical names for them to hide the middleware as shown above.

However when there is only one input, output or error/dead letter endpoint in a component we recommend using the common posix shell names **in**, **out** and **err**

3.7. Crypto (Digital Signatures)

Using Camel cryptographic endpoints and Java's Cryptographic extension it is easy to create Digital Signatures for Exchanges. Camel provides a pair of flexible endpoints which get used in concert to create a signature for an exchange in one part of the exchange's workflow and then verify the signature in a later part of the workflow.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-crypto</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.7.1. Introduction

Digital signatures make use of Asymmetric Cryptographic techniques to sign messages. From a (very) high level, the algorithms use pairs of complimentary keys with the special property that data encrypted with one key can only be decrypted with the other. One, the private key, is closely guarded and used to 'sign' the message while the other, public key, is shared around to anyone interested in verifying the signed messages. Messages are signed by using the private key to encrypting a digest of the message. This encrypted digest is transmitted along with the message. On the other side the verifier recalculates the message digest and uses the public key to decrypt the

the digest in the signature. If both digests match the verifier knows only the holder of the private key could have created the signature.

Camel uses the Signature service from the Java Cryptographic Extension to do all the heavy cryptographic lifting required to create exchange signatures.

3.7.2. URI Format

As mentioned Camel provides a pair of crypto endpoints to create and verify signatures:

```
crypto:sign:name[?options]
crypto:verify:name[?options]
```

- crypto:sign creates the signature and stores it in the Header keyed by the constant Exchange.SIGNATURE, i.e. "CamelDigitalSignature".

- crypto:sign creates the signature and stores it in the Header keyed by the constant Exchange.SIGNATURE, i.e. "CamelDigitalSignature".

In order to correctly function, the sign and verify process needs a pair of keys to be shared, signing requiring a PrivateKey and verifying a PublicKey (or a Certificate containing one). Using the JCE it is very simple to generate these key pairs but it is usually most secure to use a KeyStore to house and share your keys. The DSL is very flexible about how keys are supplied and provides a number of mechanisms.

The most basic way to way to sign an verify an exchange is with a KeyPair as follows:

```
from("direct:keypair").to("crypto:sign://basic?privateKey=#myPrivateKey",
            "crypto:verify://basic?publicKey=#myPublicKey", "mock:result");
```

The same can be achieved with the Spring XML Extensions using references to keys:

```
<route>
    <from uri="direct:keypair"/>
    <to uri="crypto:sign://basic?privateKey=#myPrivateKey"/>
    <to uri="crypto:verify://basic?publicKey=#myPublicKey"/>
    <to uri="mock:result"/>
</route>
```

See the Camel Website for the most up-to-date examples of more advanced usages of this component.

3.7.3. Options

Name	Type	Default	Description
algorithm	String	DSA	The name of the JCE Signature algorithm that will be used.
alias	String	null	An alias name that will be used to select a key from the keystore.
bufferSize	Integer	2048	The size of the buffer used in the signature process.
certificate	Certificate	null	A Certificate used to verify the signature of the exchange's payload. Either this or a Public Key is required.
keystore	KeyStore	null	A reference to a JCE Keystore that stores keys and certificates used to sign and verify.

Name	Type	Default	Description
provider	String	null	The name of the JCE Security Provider that should be used.
privateKey	PrivateKey	null	The private key used to sign the exchange's payload.
publicKey	PublicKey	null	The public key used to verify the signature of the exchange's payload.
secureRandom	secureRandom	null	A reference to a SecureRandom object that will be used to initialize the Signature service.
password	char[]	null	The password for the keystore.
clearHeaders	String	true	Remove camel crypto headers from Message after a verify operation (value can be "true"/"false").

3.8. CXF

Note

When using CXF as a consumer, the Section 3.9, "CXF Bean Component" allows you to factor out how message payloads are received from their processing as a RESTful or SOAP web service. This has the potential of using a multitude of transports to consume web services. The bean component's configuration is also simpler and provides the fastest method to implement web services using Camel and CXF.

The **cxf:** component provides integration with Apache CXF for connecting to JAX-WS services hosted in CXF.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-cxf</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

CXF dependencies

If you want to learn about CXF dependencies you can checkout the `WHICH-JARS` text file.

3.8.1. URI format

There are two scenarios:

```
cxf:bean:cxfEndpoint[?options]
```

where **cxfEndpoint** represents a bean ID that references a bean in the Spring bean registry. With this URI format, most of the endpoint details are specified in the bean definition.

```
cxf://someAddress[?options]
```

where **someAddress** specifies the CXF endpoint's address. With this URI format, most of the endpoint details are specified using options.

___

For either style above, you can append options to the URI as follows:

```
cxf:bean:cxfEndpoint?wsdlURL=wsdl/hello_world.wsdl&dataFormat=PAYLOAD
```

3.8.2. Options

Name	Required	Description
wsdlURL	No	The location of the WSDL. It is obtained from endpoint address by default. *Example:* `file://local/wsdl/hello.wsdl` or `wsdl/hello.wsdl`
serviceClass	Yes	The name of the SEI (Service Endpoint Interface) class. This class can have, but does not require, JSR181 annotations. **Since 2.0,** this option is only required by POJO mode. If the wsdlURL option is provided, serviceClass is not required for PAYLOAD and MESSAGE mode. When wsdlURL option is used without serviceClass, the serviceName and portName (endpointName for Spring configuration) options **MUST** be provided. It is possible to use # notation to reference a `serviceClass` object instance from the registry. For example, `serviceClass=#beanName`. Since 2.8, it is possible to omit both wsdlURL and serviceClass options for PAYLOAD and MESSAGE mode. When they are omitted, arbitrary XML elements can be put in CxfPayload's body in PAYLOAD mode to facilitate CXF Dispatch Mode. Please be advised that the **referenced object cannot be a Proxy** (Spring AOP Proxy is OK) as it relies on `Object.getClass().getName()` method for non Spring AOP Proxy. *Example:* `org.apache.camel.Hello`
serviceClassInstance	Yes	Use either `serviceClass` or `serviceClassInstance`. *Deprecated in 2.x.* In 1.6.x `serviceClassInstance` works like `serviceClass=#beanName`, which looks up a `serviceObject` instance from the registry. *Example:* `serviceClassInstance= beanName`
serviceName	No*	The service name this service is implementing, it maps to the `wsdl:service@name` *Required* for camel-cxf consumer since camel-2.2.0 or if more than one serviceName is present in WSDL. *Example:* {http:-//org.apache.camel}ServiceName
portName	No*	The port name this service is implementing, it maps to the `wsdl:port@name` *Required* for camel-cxf consumer since camel-2.2.0 or if more than one portName is present under serviceName. *Example:* {http:-//org.apache.camel}PortName

Name	Required	Description
dataFormat	No	The data type messages supported by the CXF endpoint. *Default:* POJO *Example:* `POJO,PAYLOAD,MESSAGE`
relayHeaders	No	Please see the **Description of** `relayHeaders` **option** section for this option in 2.0. Should a CXF endpoint relay headers along the route. Currently only available when `dataFormat=POJO` *Default:* true *Example:*`true,false`
wrapped	No	Which kind of operation that CXF endpoint producer will invoke. *Default:*`false` *Example:*`true,false`
wrappedStyle	No	**New in 2.5.0** The WSDL style that describes how parameters are represented in the SOAP body. If the value is false, CXF will chose the document-literal unwrapped style. If the value is true, CXF will chose the document-literal wrapped style. *Default:* `Null` *Example:*`true,false`
setDefaultBus	No	This will set the default bus when CXF endpoint create a bus by itself. *Default:* `false` *Example:*`true,false`
bus	No	New in 2.0. A default bus created by CXF Bus Factory. Use # notation to reference a bus object from the registry. The referenced object must be an instance of `org.apache.cxf.Bus`. *Example:* `bus=#busName`
cxfBinding	No	New in 2.0, use # notation to reference a CXF binding object from the registry. The referenced object must be an instance of `org.apache.camel.component.cxf.CxfBinding` (use an instance of `org.apache.camel.component.cxf.DefaultCxfBinding`). *Example:* `cxfBinding=#bindingName`
headerFilterStrategy	No	New in 2.0, use # notation to reference a header filter strategy object from the registry. The referenced object must be an instance of `org.apache.camel.spi.HeaderFilterStrategy` (use an instance of `org.apache.camel.component.cxf.CxfHeaderFilter-Strategy`).. *Example:* `headerFilterStrategy=#strategyName`

Name	Required	Description
loggingFeatureEnabled	No	New in 2.3, this option enables CXF Logging Feature which writes inbound and outbound SOAP messages to log. *Default:*false *Example:*loggingFeatureEnabled=true
defaultOperationName	No	New in 2.4, this option will set the default operationName that will be used by the CxfProducer which invokes the remote service. *Default:* null *Example:*defaultOperationName=greetMe
defaultOperationName-Space	No	New in 2.4, this option will set the default operationNamespace that will be used by the CxfProducer which invokes the remote service. *Default:* null *Example:*defaultOperationNamespace= http://apache.org/hello_world_soap_http
synchronous	No	New in 2.5, this option will let cxf endpoint decide to use sync or async API to do the underlying work. The default value is false which means camel-cxf endpoint will try to use async API by default. *Default:* false *Example:* synchronous=true
publishedEndpointUrl	No	New in 2.5, this option can override the endpointUrl that published from the WSDL which can be accessed with service address url plus ?wsdl. *Default:* null *Example:* publshedEndpointUrl=http://example.com/service

The `serviceName` and `portName` are QNames, so if you provide them, be sure to prefix them with their {namespace} as shown in the examples above. **NOTE:** the `serviceClass` for a CXF producer (that is, the `to` endpoint) should be a Java interface.

3.8.2.1. The descriptions of the dataformats

DataFormat	Description
POJO	POJOs (Plain old Java objects) are the Java parameters to the method being invoked on the target server. Both Protocol and Logical JAX-WS handlers are supported.
PAYLOAD	PAYLOAD is the message payload (the contents of the `soap:body`) after message configuration in the CXF endpoint is applied. Only Protocol JAX-WS handler is supported. Logical JAX-WS handler is not supported.

DataFormat	Description
MESSAGE	MESSAGE is the raw message that is received from the transport layer. JAX-WS handler is not supported.

You can determine the data format mode of an exchange by retrieving the exchange property, `CamelCXFDataFormat`. The exchange key constant is defined in `org.apache.camel.component.cxf.CxfConstants.DATA_FORMAT_PROPERTY`.

3.8.2.2. How to enable CXF's LoggingOutInterceptor in MESSAGE mode

CXF's `LoggingOutInterceptor` outputs outbound message that goes on the wire to logging system (Java Util Logging). Since the `LoggingOutInterceptor` is in `PRE_STREAM` phase (but `PRE_STREAM` phase is removed in `MESSAGE` mode), you have to configure `LoggingOutInterceptor` to be run during the `WRITE` phase. The following is an example:

```
<bean id="loggingOutInterceptor"
             class="org.apache.cxf.interceptor.LoggingOutInterceptor">
    <!-- it really should have been user-prestream, -->
    <!-- but CXF does have such phase! -->
    <constructor-arg value="write"/>
</bean>

<cxf:cxfEndpoint id="serviceEndpoint"
        address="http://localhost:9002/helloworld"
        serviceClass="org.apache.camel.component.cxf.HelloService">
    <cxf:outInterceptors>
        <ref bean="loggingOutInterceptor"/>
    </cxf:outInterceptors>
    <cxf:properties>
        <entry key="dataFormat" value="MESSAGE"/>
    </cxf:properties>
</cxf:cxfEndpoint>
```

3.8.2.3. Description of relayHeaders option

There are *in-band* and *out-of-band* on-the-wire headers from the perspective of a JAXWS WSDL-first developer.

The *in-band* headers are headers that are explicitly defined as part of the WSDL binding contract for an endpoint such as SOAP headers.

The *out-of-band* headers are headers that are serialized over the wire, but are not explicitly part of the WSDL binding contract.

Headers relaying/filtering is bi-directional.

When a route has a CXF endpoint and the developer needs to have on-the-wire headers, such as SOAP headers, be relayed along the route to be consumed say by another JAXWS endpoint, then `relayHeaders` should be set to `true`, which is the default value.

The `relayHeaders=true` express an intent to relay the headers. The decision on whether a given header is relayed is delegated to a pluggable instance that implements the `MessageHeadersRelay` interface. A concrete implementation of `MessageHeadersRelay` will be consulted to decide if a header needs to be relayed or not.

There is already an implementation of `SoapMessageHeadersRelay` which binds itself to well-known SOAP name spaces. Currently only out-of-band headers are filtered, and in-band headers will always be relayed when `relayHeaders=true` . If there is a header on the wire, whose name space is unknown to the runtime, then a fall back `DefaultMessageHeadersRelay` will be used, which simply allows all headers to be relayed.

The `relayHeaders=false` setting asserts that all headers in-band and out-of-band will be dropped.

- `POJO` and `PAYLOAD` modes are supported. In `POJO` mode, only out-of-band message headers are available for filtering as the in-band headers have been processed and removed from header list by CXF. The in-band headers are incorporated into the `MessageContentList` in POJO mode. If filtering of in-band headers is required, please use `PAYLOAD` mode or plug in a (pretty straightforward) CXF interceptor/JAXWS Handler to the CXF endpoint.

- The Message Header Relay mechanism has been merged into `CxfHeaderFilterStrategy` . The `relayHeaders` option, its semantics, and default value remain the same, but it is a property of `CxfHeaderFilterStrategy` . Here is an example of configuring it:

```
<bean id="dropAllMessageHeadersStrategy"
    class="org.apache.camel.component.cxf.CxfHeaderFilterStrategy">
     <!--  Set relayHeaders to false to drop all SOAP headers -->
     <property name="relayHeaders" value="false"/>

</bean>
```

Then, your endpoint can reference the `CxfHeaderFilterStrategy` .

```
<route>
    <from uri="cxf:bean:routerNoRelayEndpoint?headerFilterStrategy
=#dropAllMessageHeadersStrategy"/>
    <to uri="cxf:bean:serviceNoRelayEndpoint?headerFilterStrategy
=#dropAllMessageHeadersStrategy"/>
</route>
```

- The `MessageHeadersRelay` interface has changed slightly and has been renamed to `MessageHeaderFilter` . It is a property of `CxfHeaderFilterStrategy` . Here is an example of configuring user defined Message Header Filters:

```
<bean id="customMessageFilterStrategy"
    class="org.apache.camel.component.cxf.CxfHeaderFilterStrategy">
    <property name="messageHeaderFilters">
        <list>
           <!-- SoapMessageHeaderFilter is the built in filter.  -->
           <!-- It can be removed by omitting it. -->
           <bean class=
               "org.apache.camel.component.cxf.SoapMessageHeaderFilter"/>

           <!--  Add custom filter here -->
           <bean class=
               "org.apache.camel.component.cxf.soap.CustomHeaderFilter"/>
        </list>
    </property>
</bean>
```

- Other than `relayHeaders`, there are new properties that can be configured in `CxfHeaderFilterStrategy`.

Name	Description	type	Required?	Default value
relayHeaders	All message headers will be processed by Message Header Filters	boolean	No	true
relayAll-MessageHeaders	All message headers will be propagated (without processing by Message Header Filters)	boolean	No	false
allowFilter-NamespaceClash	If two filters overlap in activation namespace, the property control how it should be handled. If the value is true, last one wins. If the value is false, it will throw an exception	boolean	No	false

3.8.3. Configure the CXF endpoints with Spring

You can configure the CXF endpoint with the Spring configuration file shown below, and you can also embed the endpoint into the camelContext tags. When you are invoking the service endpoint, you can set the operationName and operationNamespace headers to explicitly state which operation you are calling.

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:cxf="http://camel.apache.org/schema/cxf"
        xsi:schemaLocation="
http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://camel.apache.org/schema/cxf
        http://camel.apache.org/schema/cxf/camel-cxf.xsd
http://camel.apache.org/schema/spring
        http://camel.apache.org/schema/spring/camel-spring.xsd     ">

  <cxf:cxfEndpoint id="routerEndpoint"
        address="http://localhost:9003/CamelContext/RouterPort"
        serviceClass="org.apache.hello_world_soap_http.GreeterImpl"/>

  <cxf:cxfEndpoint id="serviceEndpoint"
        address="http://localhost:9000/SoapContext/SoapPort"
        wsdlURL="testutils/hello_world.wsdl"
        serviceClass="org.apache.hello_world_soap_http.Greeter"
        endpointName="s:SoapPort"
        serviceName="s:SOAPService"
        xmlns:s="http://apache.org/hello_world_soap_http" />

   <camelContext id="camel"
        xmlns="http://activemq.apache.org/camel/schema/spring">
        <route>
            <from uri="cxf:bean:routerEndpoint" />
            <to uri="cxf:bean:serviceEndpoint" />
        </route>
   </camelContext>
</beans>
```

Be sure to include the JAX-WS schemaLocation attribute specified on the root beans element. This allows CXF to validate the file and is required. Also note the namespace declarations at the end of the

`<cxf:cxfEndpoint/>` tag--these are required because the combined `{namespace}localName` syntax is presently not supported for this tag's attribute values.

The `cxf:cxfEndpoint` element supports many additional attributes:

Name	Value
PortName	The endpoint name this service is implementing, it maps to the `wsdl:port@name`. In the format of `ns:PORT_NAME` where `ns` is a namespace prefix valid at this scope.
serviceName	The service name this service is implementing, it maps to the `wsdl:service@name`. In the format of `ns:SERVICE_NAME` where `ns` is a namespace prefix valid at this scope.
wsdlURL	The location of the WSDL. Can be on the classpath, file system, or be hosted remotely.
bindingId	The `bindingId` for the service model to use.
address	The service publish address.
bus	The bus name that will be used in the JAX-WS endpoint.
serviceClass	The class name of the SEI (Service Endpoint Interface) class which could have JSR181 annotation or not.

It also supports many child elements:

Name	Value
cxf:inInterceptors	The incoming interceptors for this endpoint. A list of `<bean>` or `<ref>`.
cxf:inFaultInterceptors	The incoming fault interceptors for this endpoint. A list of `<bean>` or `<ref>`.
cxf:outInterceptors	The outgoing interceptors for this endpoint. A list of `<bean>` or `<ref>`.
cxf:outFaultInterceptors	The outgoing fault interceptors for this endpoint. A list of `<bean>` or `<ref>`.
cxf:properties	A properties map which should be supplied to the JAX-WS endpoint. See below.
cxf:handlers	A JAX-WS handler list which should be supplied to the JAX-WS endpoint. See below.
cxf:dataBinding	You can specify the which `DataBinding` will be use in the endpoint. This can be supplied using the Spring `<bean class="MyDataBinding"/>` syntax.
cxf:binding	You can specify the `BindingFactory` for this endpoint to use. This can be supplied using the Spring `<bean class="MyBindingFactory"/>` syntax.
cxf:features	The features that hold the interceptors for this endpoint. A list of {{<bean>}}s or {{<ref>}}s
cxf:schemaLocations	The schema locations for endpoint to use. A list of {{<schemaLocation>}}s
cxf:serviceFactory	The service factory for this endpoint to use. This can be supplied using the Spring `<bean class="MyServiceFactory"/>` syntax

You can find more advanced examples which show how to provide interceptors , properties and handlers here.

NOTE You can use cxf:properties to set the camel-cxf endpoint's dataFormat and setDefaultBus properties from Spring configuration file.

```
<cxf:cxfEndpoint id="testEndpoint" address="http://localhost:9000/router"
    serviceClass="org.apache.camel.component.cxf.HelloService"
```

```
      endpointName="s:PortName"
      serviceName="s:ServiceName"
      xmlns:s="http://www.example.com/test">
      <cxf:properties>
        <entry key="dataFormat" value="MESSAGE"/>
        <entry key="setDefaultBus" value="true"/>
      </cxf:properties>
    </cxf:cxfEndpoint>
```

3.8.4. How to make the camel-cxf component use log4j instead of java.util.logging

CXF's default logger is `java.util.logging`. If you want to change it to log4j, proceed as follows. Create a file, in the classpath, named `META-INF/cxf/org.apache.cxf.logger`. This file should contain the fully-qualified name of the class, `org.apache.cxf.common.logging.Log4jLogger`, with no comments, on a single line.

3.8.5. How to let camel-cxf response message with xml start document

If you are using some soap client such as PHP, you will get this kind of error, because CXF doesn't add the XML start document "<?xml version="1.0" encoding="utf-8"?>"

```
Error:sendSms: SoapFault exception:
    [Client] looks like we got no XML document in [...]
```

To resolve this issue, you need to tell StaxOutInterceptor to write the XML start document for you.

```
public class WriteXmlDeclarationInterceptor
        extends AbstractPhaseInterceptor<SoapMessage> {
  public WriteXmlDeclarationInterceptor() {
     super(Phase.PRE_STREAM);
     addBefore(StaxOutInterceptor.class.getName());
  }

  public void handleMessage(SoapMessage message) throws Fault {
     message.put("org.apache.cxf.stax.force-start-document", Boolean.TRUE);
  }
}
```

You can add a customer interceptor like this and configure it into your camel-cxf endpoint

```
<cxf:cxfEndpoint id="routerEndpoint"
      address="http://localhost:9003/CamelContext/RouterPort"
      serviceClass="org.apache.hello_world_soap_http.GreeterImpl">
   <cxf:outInterceptors>
      <!-- This interceptor will force the CXF server send the XML start -->
      <!-- document to client -->
      <bean class=
        "org.apache.camel.component.cxf.WriteXmlDeclarationInterceptor"/>
```

```
        </cxf:outInterceptors>

        <cxf:properties>
            <!-- Set the publishedEndpointUrl to override the -->
            <!-- service address from generated WSDL as you want -->
         <entry key="publishedEndpointUrl"
                        value="http://www.simple.com/services/test" />
        </cxf:properties>
</cxf:cxfEndpoint>
```

Or adding a message header for it like this:

```
// set up the response context which force start document
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("org.apache.cxf.stax.force-start-document", Boolean.TRUE);
    exchange.getOut().setHeader(Client.RESPONSE_CONTEXT, map);
```

3.8.6. How to consume a message from a camel-cxf endpoint in POJO data format

The `camel-cxf` endpoint consumer POJO data format is based on the cxf invoker, so the message header has a property with the name of `CxfConstants.OPERATION_NAME` and the message body is a list of the SEI method parameters.

```
public class PersonProcessor implements Processor {

    private static final transient Logger LOG =
            LoggerFactory.getLogger(PersonProcessor.class);

    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        LOG.info("processing exchange in camel");

        BindingOperationInfo boi = (BindingOperationInfo)exchange.getProperty(
                BindingOperationInfo.class.toString());
        if (boi != null) {
            LOG.info("boi.isUnwrapped" + boi.isUnwrapped());
        }
        // Get the parameters list which element is the holder.
        MessageContentsList msgList = (
                MessageContentsList)exchange.getIn().getBody();

        Holder<String> personId = (Holder<String>)msgList.get(0);
        Holder<String> ssn = (Holder<String>)msgList.get(1);
        Holder<String> name = (Holder<String>)msgList.get(2);

        if (personId.value == null || personId.value.length() == 0) {
            LOG.info("person id 123, so throwing exception");
            // Try to throw out the soap fault message
            org.apache.camel.wsdl_first.types.UnknownPersonFault personFault =
                    new org.apache.camel.wsdl_first.types.UnknownPersonFault();
            personFault.setPersonId("");
            org.apache.camel.wsdl_first.UnknownPersonFault fault =
```

_____

```
              new org.apache.camel.wsdl_first.UnknownPersonFault(
                  "Get the null value of person name", personFault);
          // Since Camel has its own exception handler framework, we can't
          // throw the exception to trigger it. We set the fault message
          // in the exchange for camel-cxf component handling and return
          exchange.getOut().setFault(true);
          exchange.getOut().setBody(fault);
          return;
      }

      name.value = "Bonjour";
      ssn.value = "123";
      LOG.info("setting Bonjour as the response");
      // Set the response message, first element is the return value of
      // the operation, the others are the holders of method parameters
      exchange.getOut().setBody(new Object[] {null, personId, ssn, name});
   }
}
```

3.8.7. How to prepare the message for the camel-cxf endpoint in POJO data format

The `camel-cxf` endpoint producer is based on the cxf client API. First you need to specify the operation name in the message header, then add the method parameters to a list, and initialize the message with this parameter list. The response message's body is a messageContentsList; you can get the result from that list.

Note: the message body is a `MessageContentsList`. If you want to get the object array from the message body, you can get the body using `message.getbody(Object[].class)`, as follows:

```
Exchange senderExchange = new DefaultExchange(context, ExchangePattern.InOut);
final List<String> params = new ArrayList<String>();

// Prepare the request message for the camel-cxf procedure
params.add(TEST_MESSAGE);
senderExchange.getIn().setBody(params);
senderExchange.getIn().setHeader(CxfConstants.OPERATION_NAME, ECHO_OPERATION);

Exchange exchange = template.send("direct:EndpointA", senderExchange);

org.apache.camel.Message out = exchange.getOut();

// The response message's body is a MessageContentsList whose first element
// is the return value of the operation. If there are some holder parameters,
// the holder parameter will be filled in the rest of List. The result will
// be extracted from the MessageContentsList with the String class type
MessageContentsList result = (MessageContentsList)out.getBody();
LOG.info("Received output text: " + result.get(0));
Map<String, Object> responseContext =
        CastUtils.cast((Map)out.getHeader(Client.RESPONSE_CONTEXT));
assertNotNull(responseContext);
assertEquals("We should get the response context here", "UTF-8",
        responseContext.get(org.apache.cxf.message.Message.ENCODING));
assertEquals("Reply body on Camel is wrong", "echo " +
        TEST_MESSAGE, result.get(0));
```

3.8.8. How to deal with the message for a camel-cxf endpoint in PAYLOAD data format

PAYLOAD means that you process the payload message from the SOAP envelope. You can use the Header.HEADER_LIST as the key to set or get the SOAP headers and use the List<Element> to set or get SOAP body elements.

We use the common Camel DefaultMessageImpl underlayer. Message.getBody() will return an org.apache.camel.component.cxf.CxfPayload object, which has getters for SOAP message headers and Body elements. This change enables decoupling the native CXF message from the Camel message.

```
protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {
            from(SIMPLE_ENDPOINT_URI + "&dataFormat=PAYLOAD")
                    .to("log:info").process(new Processor() {
                @SuppressWarnings("unchecked")
                public void process(final Exchange exchange) throws Exception {
                    CxfPayload<SoapHeader> requestPayload =
                            exchange.getIn().getBody(CxfPayload.class);
                    List<Element> inElements = requestPayload.getBody();
                    List<Element> outElements = new ArrayList<Element>();
                    // You can use a customer toStringConverter to turn a
                    // CxfPayLoad message into String as you want
                    String request = exchange.getIn().getBody(String.class);
                    XmlConverter converter = new XmlConverter();
                    String documentString = ECHO_RESPONSE;
                    if (inElements.get(0).getLocalName().equals("echoBoolean")) {
                        documentString = ECHO_BOOLEAN_RESPONSE;
                        assertEquals("Get a wrong request",
                                ECHO_BOOLEAN_REQUEST, request);
                    } else {
                        assertEquals("Get a wrong request", ECHO_REQUEST, request);
                    }
                    Document outDocument = converter.toDOMDocument(documentString);
                    outElements.add(outDocument.getDocumentElement());
                    // set the payload header with null
                    CxfPayload<SoapHeader> responsePayload =
                            new CxfPayload<SoapHeader>(null, outElements);
                    exchange.getOut().setBody(responsePayload);
                }
            });
        }
    };
}
```

3.8.9. How to get and set SOAP headers in POJO mode

POJO means that the data format is a "list of Java objects" when the Camel-cxf endpoint produces or consumes Camel exchanges. Even though Camel expose message body as POJOs in this mode, Camel-cxf still provides access to read and write SOAP headers. However, since CXF interceptors remove in-band SOAP headers from Header list after they have been processed, only out-of-band SOAP headers are available to Camel-cxf in POJO mode.

The following example illustrate how to get/set SOAP headers. Suppose we have a route that forwards from one Camel-cxf endpoint to another. That is, SOAP Client -> Camel -> CXF service. We can attach two processors to obtain/insert SOAP headers at (1) before request goes out to the CXF service and (2) before response comes back to the SOAP Client. Processor (1) and (2) in this example are InsertRequestOutHeaderProcessor and InsertResponseOutHeaderProcessor. Our route looks like this:

```
<route>
    <from uri="cxf:bean:routerRelayEndpointWithInsertion"/>
    <process ref="InsertRequestOutHeaderProcessor" />
    <to uri="cxf:bean:serviceRelayEndpointWithInsertion"/>
    <process ref="InsertResponseOutHeaderProcessor" />
</route>
```

In 2.x SOAP headers are propagated to and from Camel Message headers. The Camel message header name is "org.apache.cxf.headers.Header.list" which is a constant defined in CXF (org.apache.cxf.headers.Header.HEADER_LIST). The header value is a List of CXF SoapHeader objects (org.apache.cxf.binding.soap.SoapHeader). The following snippet is the InsertResponseOutHeaderProcessor (that inserts a new SOAP header in the response message). The way to access SOAP headers in both InsertResponseOutHeaderProcessor and InsertRequestOutHeaderProcessor is the same. The only difference between the two processors is setting the direction of the inserted SOAP header.

```
public static class InsertResponseOutHeaderProcessor implements Processor {

    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        List<SoapHeader> soapHeaders =
                (List)exchange.getIn().getHeader(Header.HEADER_LIST);

        // Insert a new header
        String xml =
                "<?xml version=\"1.0\" encoding=\"utf-8\"?><outofbandHeader "
            + "xmlns=\"http://cxf.apache.org/outofband/Header\" "
            + "hdrAttribute=\"testHdrAttribute\" "
            + "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\" "
            + "soap:mustUnderstand=\"1\"><name>"
            + "New_testOobHeader</name><value>New_testOobHeaderValue"
            + "</value></outofbandHeader>";

        SoapHeader newHeader = new SoapHeader(soapHeaders.get(0).getName(),
                DOMUtils.readXml(new StringReader(xml)).
                    getDocumentElement());

        // make sure direction is OUT since it is a response message.
        newHeader.setDirection(Direction.DIRECTION_OUT);
        //newHeader.setMustUnderstand(false);
        soapHeaders.add(newHeader);

    }

}
```

In 1.x SOAP headers are not propagated to and from Camel Message headers. Users have to go deeper into CXF APIs to access SOAP headers. Also, accessing the SOAP headers in a request message is slight different than in a response message. The InsertRequestOutHeaderProcessor and InsertResponseOutHeaderProcessor are as follows:

```
public static class InsertRequestOutHeaderProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
```

```
        CxfMessage message = exchange.getIn().getBody(CxfMessage.class);
        Message cxf = message.getMessage();
        List<SoapHeader> soapHeaders = (List)cxf.get(Header.HEADER_LIST);

        // Insert a new header
        String xml =
            "<?xml version=\"1.0\" encoding=\"utf-8\"?><outofbandHeader "
            + "xmlns=\"http://cxf.apache.org/outofband/Header\" "
            + "hdrAttribute=\"testHdrAttribute\" "
            + "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\""
            + " soap:mustUnderstand=\"1\"><name>"
            + "New_testOobHeader</name><value>New_testOobHeaderValue"
            + "</value></outofbandHeader>";

        SoapHeader newHeader = new SoapHeader(soapHeaders.get(0).getName(),
            DOMUtils.readXml(new StringReader(xml)).getDocumentElement());

        // make sure direction is IN since it is a request message.
        newHeader.setDirection(Direction.DIRECTION_IN);
        //newHeader.setMustUnderstand(false);
        soapHeaders.add(newHeader);
    }
}

public static class InsertResponseOutHeaderProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        CxfMessage message = exchange.getIn().getBody(CxfMessage.class);
        Map responseContext =
                (Map)message.getMessage().get(Client.RESPONSE_CONTEXT);
        List<SoapHeader> soapHeaders =
                (List)responseContext.get(Header.HEADER_LIST);

        // Insert a new header
        String xml = "<?xml version=\"1.0\" encoding=\"utf-8\"?>"
            + "<outofbandHeader xmlns="
            + "\"http://cxf.apache.org/outofband/Header\" "
            + "hdrAttribute=\"testHdrAttribute\" "
            + "xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\" "
            + "soap:mustUnderstand=\"1\">"
            + "<name>New_testOobHeader</name><value>"
            + "New_testOobHeaderValue</value></outofbandHeader>";

        SoapHeader newHeader = new SoapHeader(soapHeaders.get(0).getName(),
            DOMUtils.readXml(new StringReader(xml)).getDocumentElement()
        );

        // make sure direction is OUT since it is a response message.
        newHeader.setDirection(Direction.DIRECTION_OUT);
        //newHeader.setMustUnderstand(false);
        soapHeaders.add(newHeader);

    }
}
```

3.8.10. How to get and set SOAP headers in PAYLOAD mode

We've already shown how to access SOAP message (CxfPayload object) in PAYLOAD mode (See "How to deal with the message for a camel-cxf endpoint in PAYLOAD data format").

In 2.x Once you obtain a CxfPayload object, you can invoke the CxfPayload.getHeaders() method that returns a List of DOM Elements (SOAP headers).

```
from(getRouterEndpointURI()).process(new Processor() {
    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        CxfPayload<SoapHeader> payload =
                exchange.getIn().getBody(CxfPayload.class);
        List<Element> elements = payload.getBody();
        assertNotNull("We should get the elements here", elements);
        assertEquals("Get the wrong elements size", 1, elements.size());
        assertEquals("Get the wrong namespace URI",
                "http://camel.apache.org/pizza/types",
                elements.get(0).getNamespaceURI());

        List<SoapHeader> headers = payload.getHeaders();
        assertNotNull("We should get the headers here", headers);
        assertEquals("Get the wrong headers size", headers.size(), 1);
        assertEquals("Get the wrong namespace URI",
                ((Element)(headers.get(0).getObject())).getNamespaceURI(),
                "http://camel.apache.org/pizza/types");
    }
})
.to(getServiceEndpointURI());
```

3.8.11. SOAP headers are not available in MESSAGE mode

SOAP headers are not available in MESSAGE mode as SOAP processing is skipped.

3.8.12. How to throw a SOAP Fault from Camel

If you are using a `camel-cxf` endpoint to consume the SOAP request, you may need to throw the SOAP Fault from the Camel context. Basically, you can use the `throwFault` DSL to do that; it works for `POJO`, `PAYLOAD` and `MESSAGE` data format. You can define the soap fault like this

```
SOAP_FAULT = new SoapFault(EXCEPTION_MESSAGE, SoapFault.FAULT_CODE_CLIENT);
Element detail = SOAP_FAULT.getOrCreateDetail();
Document doc = detail.getOwnerDocument();
Text tn = doc.createTextNode(DETAIL_TEXT);
detail.appendChild(tn);
```

Then throw it as you like:

```
from(routerEndpointURI).setFaultBody(constant(SOAP_FAULT));
```

If your CXF endpoint is working in the `MESSAGE` data format, you could set the SOAP Fault message in the message body and set the response code in the message header.

```
from(routerEndpointURI).process(new Processor() {

    public void process(Exchange exchange) throws Exception {
        Message out = exchange.getOut();
        // Set the message body with the
        out.setBody(this.getClass().
              getResourceAsStream("SoapFaultMessage.xml"));
        // Set the response code here
        out.setHeader(
              org.apache.cxf.message.Message.RESPONSE_CODE, new Integer(500));
    }

});
```

Same for using POJO data format. You can set the SOAPFault on the out body and also indicate it is a fault by calling Message.setFault(true):

```
from("direct:start").onException(SoapFault.class)
        .maximumRedeliveries(0).handled(true)
        .process(new Processor() {
            public void process(Exchange exchange) throws Exception {
                SoapFault fault = exchange
                  .getProperty(Exchange.EXCEPTION_CAUGHT, SoapFault.class);
                exchange.getOut().setFault(true);
                exchange.getOut().setBody(fault);
            }
        }
    ).end().to(SERVICE_URI);
```

3.8.13. How to propagate a camel-cxf endpoint's request and response context

cxf client API provides a way to invoke the operation with request and response context. If you are using a `camel-cxf` endpoint producer to invoke the outside web service, you can set the request context and get response context with the following code:

```
CxfExchange exchange =
        (CxfExchange)template.send(getJaxwsEndpointUri(), new Processor() {
   public void process(final Exchange exchange) {
      final List<String> params = new ArrayList<String>();
      params.add(TEST_MESSAGE);
      // Set the request context to the inMessage
      Map<String, Object> requestContext =
              new HashMap<String, Object>();
      requestContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
              JAXWS_SERVER_ADDRESS);
      exchange.getIn().setBody(params);
      exchange.getIn().setHeader(Client.REQUEST_CONTEXT , requestContext);
      exchange.getIn().setHeader(
              CxfConstants.OPERATION_NAME, GREET_ME_OPERATION);
   }
```

```
});

    org.apache.camel.Message out = exchange.getOut();
    // The output is an object array,
    // the first element of the array is the return value
    Object\[\] output = out.getBody(Object\[\].class);
    LOG.info("Received output text: " + output\[0\]);
    // Get the response context form outMessage
    Map<String, Object> responseContext =
        CastUtils.cast((Map)out.getHeader(Client.RESPONSE_CONTEXT));
    assertNotNull(responseContext);
    assertEquals("Get the wrong wsdl opertion name",
        "{http://apache.org/hello_world_soap_http}greetMe",
        responseContext.get("javax.xml.ws.wsdl.operation").toString());
```

3.8.14. Attachment Support

POJO Mode: Both SOAP with Attachment and MTOM are supported (see example in Payload Mode for enabling MTOM). However, SOAP with Attachment is not tested. Since attachments are marshalled and unmarshalled into POJOs, users typically do not need to deal with the attachment themselves. Attachments are propagated to Camel message's attachments since 2.1. So, it is possible to retrieve attachments by Camel Message API

```
DataHandler Message.getAttachment(String id)
```

.

Payload Mode: MTOM is supported since 2.1. Attachments can be retrieved by Camel Message APIs mentioned above. SOAP with Attachment (SwA) is supported and attachments can be retrieved since 2.5. SwA is the default (same as setting the CXF endpoint property "mtom_enabled" to false).

To enable MTOM, set the CXF endpoint property "mtom_enabled" to *true* . (I believe you can only do it with Spring.)

```
<cxf:cxfEndpoint id="routerEndpoint"
            address="http://localhost:9091/jaxws-mtom/hello"
    wsdlURL="mtom.wsdl"
    serviceName="ns:HelloService"
    endpointName="ns:HelloPort"
    xmlns:ns="http://apache.org/camel/cxf/mtom_feature">

    <cxf:properties>
    <!--  enable mtom by setting this property to true -->
    <entry key="mtom-enabled" value="true"/>

    <!--  set the camel-cxf endpoint data fromat to PAYLOAD mode -->
    <entry key="dataFormat" value="PAYLOAD"/>
</cxf:properties>
```

You can produce a Camel message with attachment to send to a CXF endpoint in Payload mode.

```
Exchange exchange = context.createProducerTemplate().send(
            "direct:testEndpoint", new Processor() {

    public void process(Exchange exchange) throws Exception {
```

```
        exchange.setPattern(ExchangePattern.InOut);
        List<Element> elements = new ArrayList<Element>();

        elements.add(DOMUtils.readXml(
            new StringReader(MtomTestHelper.REQ_MESSAGE)).getDocumentElement());
        CxfPayload<SoapHeader> body = new CxfPayload<SoapHeader>(
            new ArrayList<SoapHeader>(),elements);

        exchange.getIn().setBody(body);
        exchange.getIn().addAttachment(MtomTestHelper.REQ_PHOTO_CID,
            new DataHandler(new ByteArrayDataSource(
                MtomTestHelper.REQ_PHOTO_DATA, "application/octet-stream")));

        exchange.getIn().addAttachment(MtomTestHelper.REQ_IMAGE_CID,
            new DataHandler(new ByteArrayDataSource(
                MtomTestHelper.requestJpeg, "image/jpeg")));
    }
});

// process response

CxfPayload<SoapHeader> out = exchange.getOut().getBody(CxfPayload.class);
Assert.assertEquals(1, out.getBody().size());

Map<String, String> ns = new HashMap<String, String>();
ns.put("ns", MtomTestHelper.SERVICE_TYPES_NS);
ns.put("xop", MtomTestHelper.XOP_NS);

XPathUtils xu = new XPathUtils(ns);
Element ele = (Element)xu.getValue(
            "//ns:DetailResponse/ns:photo/xop:Include",
            out.getBody().get(0),XPathConstants.NODE);

String photoId = ele.getAttribute("href").substring(4); // skip "cid:"

ele = (Element)xu.getValue(
            "//ns:DetailResponse/ns:image/xop:Include",
            out.getBody().get(0),XPathConstants.NODE);

String imageId = ele.getAttribute("href").substring(4); // skip "cid:"

DataHandler dr = exchange.getOut().getAttachment(photoId);
Assert.assertEquals("application/octet-stream", dr.getContentType());
MtomTestHelper.assertEquals(
            MtomTestHelper.RESP_PHOTO_DATA,
            IOUtils.readBytesFromStream(dr.getInputStream()));

dr = exchange.getOut().getAttachment(imageId);
Assert.assertEquals("image/jpeg", dr.getContentType());

BufferedImage image = ImageIO.read(dr.getInputStream());
Assert.assertEquals(560, image.getWidth());
Assert.assertEquals(300, image.getHeight());
```

You can also consume a Camel message received from a CXF endpoint in Payload mode.

```java
public static class MyProcessor implements Processor {

    @SuppressWarnings("unchecked")
    public void process(Exchange exchange) throws Exception {
        CxfPayload<SoapHeader> in = exchange.getIn().getBody(CxfPayload.class);

        // verify request
        Assert.assertEquals(1, in.getBody().size());

        Map<String, String> ns = new HashMap<String, String>();
        ns.put("ns", MtomTestHelper.SERVICE_TYPES_NS);
        ns.put("xop", MtomTestHelper.XOP_NS);

        XPathUtils xu = new XPathUtils(ns);
        Element ele = (Element)xu.getValue("//ns:Detail/ns:photo/xop:Include",
                    in.getBody().get(0),XPathConstants.NODE);

        String photoId = ele.getAttribute("href").substring(4); // skip "cid:"
        Assert.assertEquals(MtomTestHelper.REQ_PHOTO_CID, photoId);

        ele = (Element)xu.getValue("//ns:Detail/ns:image/xop:Include",
                    in.getBody().get(0), XPathConstants.NODE);

        String imageId = ele.getAttribute("href").substring(4); // skip "cid:"
        Assert.assertEquals(MtomTestHelper.REQ_IMAGE_CID, imageId);

        DataHandler dr = exchange.getIn().getAttachment(photoId);
        Assert.assertEquals("application/octet-stream", dr.getContentType());
        MtomTestHelper.assertEquals(MtomTestHelper.REQ_PHOTO_DATA,
                    IOUtils.readBytesFromStream(dr.getInputStream()));

        dr = exchange.getIn().getAttachment(imageId);
        Assert.assertEquals("image/jpeg", dr.getContentType());
        MtomTestHelper.assertEquals(MtomTestHelper.requestJpeg,
                IOUtils.readBytesFromStream(dr.getInputStream()));

          // create response
        List<Element> elements = new ArrayList<Element>();
        elements.add(DOMUtils.readXml(new StringReader(
                MtomTestHelper.RESP_MESSAGE)).getDocumentElement());
        CxfPayload<SoapHeader> body = new CxfPayload<SoapHeader>(
                new ArrayList<SoapHeader>(),elements);
        exchange.getOut().setBody(body);
        exchange.getOut().addAttachment(MtomTestHelper.RESP_PHOTO_CID,
                new DataHandler(new ByteArrayDataSource(
                    MtomTestHelper.RESP_PHOTO_DATA, "application/octet-stream")));

        exchange.getOut().addAttachment(MtomTestHelper.RESP_IMAGE_CID,
                new DataHandler(new ByteArrayDataSource(
                    MtomTestHelper.responseJpeg, "image/jpeg")));

    }
}
```

Message Mode: Attachments are not supported as it does not process the message at all.

3.9. CXF Bean Component

The **cxfbean:** component allows other Camel endpoints to send exchange and invoke Web service bean objects. (**Currently, it only supports JAXRS, JAXWS(new to camel2.1) annotated service bean.**)

Note : CxfBeanEndpoint is a `ProcessorEndpoint` so it has no consumers. It works similarly to a Bean component.

3.9.1. URI format

```
cxfbean:serviceBeanRef
```

where **serviceBeanRef** is a registry key to look up the service bean object. If `serviceBeanRef` references a `List` object, elements of the `List` are the service bean objects accepted by the endpoint.

3.9.2. Options

Name	Required	Description
cxfBeanBinding	No	CXF bean binding specified by the # notation. The referenced object must be an instance of `org.apache.camel.component.cxf.cxfbean.CxfBeanBinding`. *Default:* `DefaultCxfBeanBinding` *Example:* `cxfBinding=#bindingName`
bus	No	CXF bus reference specified by the # notation. The referenced object must be an instance of `org.apache.cxf.Bus`. *Default:* `Default bus created by CXF Bus Factory` *Example:* `bus=#busName`
headerFilterStrategy	No	Header filter strategy specified by the # notation. The referenced object must be an instance of `org.apache.camel.spi.HeaderFilterStrategy`. *Default:* `CxfHeaderFilterStrategy` *Example:* `headerFilterStrategy=#strategyName`
setDefaultBus	No	This will set the default bus when CXF endpoint create a bus by itself. *Default:* `false` *Example:* `true,false`
populateFromClass	No	Since 2.3, the wsdlLocation annotated in the POJO is ignored (by default) unless this option is set to `false`. Prior to 2.3, the

Name	Required	Description
		wsdlLocation annotated in the POJO is always honored and it is not possible to ignore. *Default:* `true` *Example:* `true,false`
providers	No	Since 2.5, setting the providers for the CXFRS endpoint. *Default:* `null` *Example:* `providers=#providerRef1,#providerRef2`

3.9.3. Headers

Compatiblity

Currently, CXF Bean component has (only) been tested with Jetty HTTP component -- it can understand headers from Jetty HTTP component without requiring conversion.

Name	Required	Description
CamelHttpCharacterEncoding (before 2.0-m2: `CamelCxfBean-CharacterEncoding`)	None	Character encoding *Type:* `String` *In/Out:* In *Default:* `None` *Example:* ISO-8859-1
CamelContentType (before 2.0-m2: `CamelCxfBeanContentType`)	No	Content type *Type:* `String` *In/Out:* In *Default:* */* *Example:* `text/xml`
CamelHttpBaseUri (2.0-m3 and before: `CamelCxfBeanRequestBasePath`)	Yes	The value of this header will be set in the CXF message as the `Message.BASE_PATH` property. It is needed by CXF JAX-RS processing. Basically, it is the scheme, host and port portion of the request URI. *Type:* `String` *In/Out:* In *Default:* The Endpoint URI of the source endpoint in the Camel exchange *Example:* http://localhost:9000
CamelHttpPath (before 2.0-m2: `CamelCxfBeanRequestPath`)	Yes	Request URI's path

Name	Required	Description
		Type: `String` *In/Out:* In *Default:* None *Example:* consumer/123
`CamelHttpMethod` (before 2.0-m2: `CamelCxfBeanVerb`)	Yes	RESTful request verb *Type:* `String` *In/Out:* In *Default:* None *Example:* `GET,PUT,POST,DELETE`
`CamelHttpResponseCode`	No	HTTP response code *Type:* `Integer` *In/Out:* Out *Default:* None *Example:* 200

3.9.4. A Working Sample

This sample shows how to create a route that starts a Jetty HTTP server. The route sends requests to a CXF Bean and invokes a JAXRS annotated service.

First, create a route as follows. The `from` endpoint is a Jetty HTTP endpoint that is listening on port 9000. Notice that the `matchOnUriPrefix` option must be set to `true` because RESTful request URI will not match the endpoint's URI http:-//localhost:9000 exactly.

```
<route>
    <from uri="jetty:http://localhost:9000?matchOnUriPrefix=true" />
    <to uri="cxfbean:customerServiceBean" />
</route>
```

The `to` endpoint is a CXF Bean with bean name `customerServiceBean`. The name will be looked up from the registry. Next, we make sure our service bean is available in Spring registry. We create a bean definition in the Spring configuration. In this example, we create a List of service beans (of one element). We could have created just a single bean without a List.

```
<util:list id="customerServiceBean">
    <bean class="org.apache.camel.component.cxf.testbean.CustomerService"/>
</util:list>

<bean class="org.apache.camel.wsdl_first.PersonImpl" id="jaxwsBean" />
```

That's it. Once the route is started, the web service is ready for business. A HTTP client can make a request and receive response.

```
url = new URL(
    "http://localhost:9000/customerservice/orders/223/products/323");
```

```
in = url.openStream();
assertEquals("{\"Product\":{\"description\":\"product 323\",\"id\":323}}",
            CxfUtils.getStringFromInputStream(in));
```

3.10. CXFRS

Note

When using CXF as a consumer, the Section 3.9, "CXF Bean Component" allows you to factor out how message payloads are received from their processing as a RESTful or SOAP web service. This has the potential of using a multitude of transports to consume web services. The bean component's configuration is also simpler and provides the fastest method to implement web services using Camel and CXF.

The **cxfrs:** component provides integration with Apache CXF for connecting to JAX-RS services hosted in CXF.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-cxf</artifactId>
    <!-- use the same version as your Camel core version -->
    <version>x.x.x</version>
</dependency>
```

3.10.1. URI format

```
cxfrs://address?options
```

where **address** represents the CXF endpoint's address

```
cxfrs:bean:rsEndpoint
```

where **rsEndpoint** represents the Spring bean's name which presents the CXFRS client or server

For either style above, you can append options to the URI as follows:

```
cxfrs:bean:cxfEndpoint?resourceClasses=org.apache.camel.rs.Example
```

3.10.2. Options

Name	Required	Description
resourceClasses	No	The resource classes which you want to export as REST service. Multiple classes can be separated by comma. *Default: None* *Example:* resourceClasses= org.apache.camel.rs.Example1, org.apache.camel.rs.Exchange2

Name	Required	Description
`httpClientAPI`	No	If it is true, the CxfRsProducer will use the HttpClientAPI to invoke the service If it is false, the CxfRsProducer will use the ProxyClientAPI to invoke the service *Default: true* *Example:* httpClientAPI=true
synchronous	No	New in 2.5, this option will let CxfRsConsumer decide to use sync or async API to do the underlying work. The default value is false which means it will try to use async API by default. *Default:*false *Example:*synchronous=true
throwExceptionOnFailure	No	New in 2.6, this option tells the CxfRsProducer to inspect return codes and will generate an Exception if the return code is larger than 207. *Default:*true *Example:*throwExceptionOnFailure=true
maxClientCacheSize	No	New in 2.6, you can set a IN message header CamelDestinationOverrideUrl to dynamically override the target destination Web Service or REST Service defined in your routes. The implementation caches CXF clients or ClientFactoryBean in CxfProvider and CxfRsProvider. This option allows you to configure the maximum size of the cache. *Default:*10 *Example:*maxClientCacheSize=5
`setDefaultBus`	false	If true, will set the default bus when CXF endpoint create a bus by itself.
`bus`		A default bus created by CXF Bus Factory. Prefix bus name with a # to reference a bus object from the registry. The referenced object must be an instance of org.apache.cxf.Bus.

You can also configure the CXF REST endpoint through the Spring configuration. Since there are lots of difference between the CXF REST client and CXF REST Server, we provides different configuration for them. Please check out the schema file and CXF REST user guide for more information.

See the Camel Website for the latest examples of this component in use.

3.11. Direct

The **direct:** component provides direct, synchronous invocation of any consumers when a producer sends a message exchange. This endpoint can be used to connect existing routes in the **same** Camel context.

> ### Asynchronous
>
> The Section 3.39, "SEDA" component provides asynchronous invocation of any consumers when a producer sends a message exchange.

> ### Connection to other Camel contexts
>
> The Section 3.52, "VM" component provides connections between Camel contexts as long they run in the same **JVM** .

3.11.1. URI format

```
direct:someName[?options]
```

where **someName** can be any string to uniquely identify the endpoint.

3.11.2. Samples

In the route below we use the direct component to link the two routes together:

```
from("activemq:queue:order.in")
    .to("bean:orderServer?method=validate")
    .to("direct:processOrder");

from("direct:processOrder")
    .to("bean:orderService?method=process")
    .to("activemq:queue:order.out");
```

and the sample using Spring DSL:

```
<route>
    <from uri="activemq:queue:order.in"/>
    <to uri="bean:orderService?method=validate"/>
    <to uri="direct:processOrder"/>
</route>

<route>
    <from uri="direct:processOrder"/>
    <to uri="bean:orderService?method=process"/>
    <to uri="activemq:queue:order.out"/>
</route>
```

See also samples from the Section 3.39, "SEDA" component, how they can be used together.

3.12. Event

The **event:** component provides access to the Spring ApplicationEvent objects. This allows you to publish ApplicationEvent objects to a Spring ApplicationContext or to consume them. You can then use Enterprise Integration Patterns to process them such as Section 2.27, "Message Filter" .

3.12.1. URI format

```
spring-event://default
```

3.13. Exec

The exec component can be used to execute system commands. For this component, Maven users will need to add the following dependency to their pom.xml file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-exec</artifactId>
    <version>${camel-version}</version>
</dependency>
```

replacing ${camel-version} with the precise version used.

This component has URI format of:

```
exec://executable[?options]
```

where executable is the name, or file path, of the system command that will be executed. If executable name is used (for example, exec:java), the executable must be in the system path.

3.13.1. URI options

Name	Default value	Description
args	null	The arguments of the executable- they may be one or many whitespace-separated tokens, that can be quoted with ", for example, args="arg 1" arg2 will use two arguments arg 1 and arg2. To include the quotes, enclose them in another set of quotes; for example, args=""arg 1"" arg2 will use the arguments "arg 1" and arg2.
workingDir	null	The directory in which the command should be executed. If null, the working directory of the current process will be used.
timeout	Long.MAX_VALUE	The timeout, in milliseconds, after which the executable should be terminated. If execution has not completed within this period, the component will send a termination request.
outFile	null	The name of a file, created by the executable, that should be considered as output of the executable. If no outFile is set, the standard output (stdout) of the executable will be used instead.
binding	a DefaultExecBinding instance	A reference to an org.apache.commons. exec.ExecBinding in the Registry .

Name	Default value	Description
commandExecutor	a DefaultCommand-Executor instance	A reference to an `org.apache.commons.exec.ExecCommandExecutor` in the Registry, that customizes the command execution. The default command executor utilizes the commons-exec library. which adds a shutdown hook for every executed command.
useStderrOnEmpty-Stdout	false	A boolean which dictates when `stdin` is empty, it should fallback and use `stderr` in the Camel Message Body. This option is default `false.`

3.13.2. Message headers

The supported headers are defined in `org.apache.camel.component.exec.ExecBinding`.

Name	Message	Description
ExecBinding.EXEC_COMMAND_EXECUTABLE	in	The name of the system command that will be executed. Overrides the `executable` in the URI. *Type:* `String`
ExecBinding.EXEC_COMMAND_ARGS	in	The arguments of the executable. The arguments are used literally, no quoting is applied. Overrides existing `args` in the URI. *Type:* `java.util.List<String>`
ExecBinding.EXEC_COMMAND_ARGS	in	The arguments of the executable as a single string where each argument is whitespace separated (see `args` in URI option). The arguments are used literally, no quoting is applied. Overrides existing `args` in the URI. *Type:* `String`
ExecBinding.EXEC_COMMAND_OUT_FILE	in	The name of a file, created by the executable, that should be considered as output of the executable. Overrides existing `outFile` in the URI. *Type:* `String`
ExecBinding.EXEC_COMMAND_TIMEOUT	in	The timeout, in milliseconds, after which the executable should be terminated. Overrides existing `timeout` in the URI. *Type:* `long`
ExecBinding.EXEC_COMMAND_WORKING_DIR	in	The directory in which the command should be executed. Overrides existing `workingDir` in the URI. *Type:* `String`
ExecBinding.EXEC_EXIT_VALUE	out	The value of this header is the *exit value* of the executable. Non-zero exit values typically indicate abnormal termination. Note that the exit value is OS-dependent.

Name	Message	Description
		Type: `int`
`ExecBinding.EXEC_STDERR`	`out`	The value of this header points to the standard error stream (stderr) of the executable. If no stderr is written, the value is `null`. *Type:* `java.io.InputStream`
`ExecBinding.EXEC_USE_STDERR_ON_EMPTY_STDOUT`	`in`	Indicates when the `stdin` is empty, should we fallback and use `stderr` as the body of the Camel message. By default this option is `false`. *Type:* `boolean`

3.13.3. Message body

If the `in` message body, that the `Exec` component receives is convertible to `java.io.InputStream`, it is used to feed the input of the executable via its stdin. After the execution, the message body is the result of the execution, that is `org.apache.camel.components.exec.ExecResult` instance containing the stdout, stderr, exit value and out file. The component supports the following `ExecResult` type converters for convenience:

From	To
`ExecResult`	`java.io.InputStream`
`ExecResult`	`String`
`ExecResult`	`byte []`
`ExecResult`	`org.w3c.dom.Document`

If out file is used (the endpoint is configured with `outFile`, or there is `ExecBinding.EXEC_COMMAND_OUT_FILE` header) the converters return the content of the out file. If no out file is used, then the converters will use the stdout of the process for conversion to the target type.

For an example, the below executes `wc` (word count, Linux) to count the words in file `/usr/share/dict/words`. The word count (output) is written in the standard output stream of `wc`.

```
from("direct:exec")
.to("exec:wc?args=--words /usr/share/dict/words")
.process(new Processor() {
    public void process(Exchange exchange) throws Exception {
      // By default, the body is ExecResult instance
      assertIsInstanceOf(ExecResult.class, exchange.getIn().getBody());
      // Use the Camel Exec String type converter to convert the ExecResult
      // to String. In this case, the stdout is considered as output.
      String wordCountOutput = exchange.getIn().getBody(String.class);
      // do something with the word count
      ...
    }
});
```

3.14. File

The File component provides access to file systems. The main functionality that this facilitates is:

- files may be processed by other Camel Components. A typical pattern is that files are written to a directory (or subdirectories) by one or more components (producers). Other components (consumers) may subsequently read, process (and move or delete) these files. Consumers may generate new files based on templates or filters being applied to the existing files. Temporary subdirectories or files may be created or used by consumers or producers as part of the processing.

- messages from other components may be saved to disk, and this may also involve applying filters to the contents, logging information in the messages, and so on.

> ### Avoid reading files currently being written by another application
>
> Beware the JDK File IO API is somewhat limited in detecting whether another application is currently writing or copying a file. The implementation semantics can also vary, depending on the OS platform. This could lead to the situation where Camel thinks the file is not locked by another process and starts consuming it. You may need to check how this is implemented for your specific environment.
>
> If needed, to assist you with this issue, Camel provides different `readLock` options and a `doneFileOption` option that you can use. See also the section *Consuming files from folders where others drop files directly*.

Should you ever need to activate debugging for this component it logs at level `trace`.

3.14.1. URI format

```
file:directoryName[?options]
```

or

```
file://directoryName[?options]
```

where **directoryName** represents the underlying file directory.

You can append query options to the URI in the following format, `?option=value&option=value&...`

> ### Only directories
>
> Camel 2.0 onwards only supports endpoints configured with a starting directory. So the **directoryName** must be a directory.
>
> If you want to consume a single file only, specify the starting directory, and then use the **fileName** option, for example by setting `fileName=info.xml`.
>
> **Note:** the starting directory must not contain dynamic expressions with ${ } placeholders; again, use the `fileName` option to specify the dynamic part of the filename.

3.14.2. URI Options

3.14.2.1. Common

Name	Default Value	Description
autoCreate	true	Automatically create missing directories in the file's pathname. For the file consumer, that means creating the starting directory. For the file producer, it means creating the directory the files should be written to.
bufferSize	128kb	Write buffer, sized in bytes.
fileName	null	Use Expression such as File Language to dynamically set the filename. For consumers, it is used as a filename filter. For producers, it is used to evaluate the filename to write. If an expression is set, it take precedence over the CamelFileName header. (**Note:** The header itself can also be an Expression). The expression options support both String and Expression types. If the expression is a String type, it is **always** evaluated using the File Language. If the expression is an Expression type, the specified Expression type is used; this allows you, for instance, to use OGNL expressions. For the consumer, you can use it to filter filenames, so you can for instance consume today's file using the File Language syntax: mydata-${date:now:yyyyMMdd}.txt .
flatten	false	Flatten is used to flatten the file name path to strip any leading paths, so it is purely the file name. This allows you to consume recursively into sub-directories. However, for example, if you write the files to another directory they will be written in a (flat) single directory. Setting this to true on the producer ensures that any file name received in CamelFileName header will be stripped of any leading paths.
charset	null	this option is used to specify the encoding of the file, and Camel will set the Exchange property with Exchange.CHARSET_NAME with the value of this option.
copyAndDelete OnRenameFail	true	Whether to fallback and do a copy and delete file, in case the file could not be renamed directly. This option is not available for the [FTP|FTP2] component.

3.14.2.2. Consumer

Name	Default Value	Description
initialDelay	1000	Milliseconds before polling the file or directory starts.
delay	500	Milliseconds before the next poll of the file or directory.
useFixedDelay	true	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.

Name	Default Value	Description
runLoggingLevel	TRACE	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.
recursive	false	if it is consuming a directory, it will look for files in all the sub-directories as well.
delete	false	If true, the file will be deleted **after** it is processed
noop	false	If true, the file is not moved or deleted in any way. This option is good for readonly data, or for ETL type requirements. If noop=true, Camel will set idempotent=true as well, to avoid consuming the same files over and over again.
preMove	null	If a file is to be moved **before** processing, use Expression such as File Language to dynamically specify the target directory name. For example to move in-progress files into the order directory set this value to order.
move	.camel	If a file is to be moved **after** processing, use Expression such as File Language to dynamically set the target directory name. To move files into a .done subdirectory just enter .done.
moveFailed	null	Expression (such as File Language) used to dynamically set a different target directory when moving files after processing (configured via move setting defined above) failed. For example, to move files into a .error subdirectory use: .error. Note: When moving the files to the "fail" location Camel will handle the error and will not pick up the file again.
include	null	Is used to include files, if filename matches the regex pattern.
exclude	null	Is used to exclude files, if filename matches the regex pattern.
antInclude	null	Ant style filter inclusion, for example {{antInclude=**/*.txt}}. Multiple inclusions may be specified in comma-delimited format.
antExclude	null	Ant style filter exclusion. If both antInclude and antExclude the latter takes precedence. Multiple exclusions may be specified in comma-delimited format.
idempotent	false	Option to use the Section 2.18, "Idempotent Consumer" EIP pattern to let Camel skip already processed files. This will by default use a memory based LRUCache that holds 1000 entries. If noop=true then idempotent will be enabled as well to avoid consuming the same files over and over again.
idempotent-Repository	null	Pluggable repository as a org.apache.camel.processor.idempotent.MessageIdRepository class. This will by default use MemoryMessageIdRepository if none is specified and idempotent is true.
inProgress-Repository	memory	A pluggable in-progress repository org.apache.camel.spi.IdempotentRepository. The in-progress repository is used to account the current in-progress files being consumed. By default a memory based repository is used.
filter	null	Pluggable filter as a org.apache.camel.component.file.GenericFileFilter class. This will skip files if filter returns false in its accept() method.
sorter	null	Pluggable sorter as a java.util.Comparator<org.apache.camel.component.file.GenericFile> class.
sortBy	null	Built-in sort using the File Language. Supports nested sorts, so you can have a sort by file name and as a second group sort by modified date. See sorting section below for details.

Name	Default Value	Description
`readLock`	`marker-File`	Used by consumer, to only poll the files if it has exclusive read-lock on the file (that is, the file is not in-progress or being written). Camel will wait until the file lock is granted. This option provides the built-in strategies: `markerFile` is where Camel will create a marker file and hold a lock on the marker file. This option is **not** avail for the FTP component. `changed` is using file length/modification timestamp to detect whether the file is currently being copied or not. This will at least use 1 sec. to determine this, so this option cannot consume files as fast as the others, but can be more reliable as the JDK IO API cannot always determine whether a file is currently being used by another process. This option is **not** avail for the FTP component. `fileLock` is for using `java.nio.channels.FileLock`. This option is **not** avail for the FTP component. `rename` is for using a try to rename the file as a test if we can get exclusive read-lock. `none` is for no read locks at all.
`readLockTimeout`	10000	Optional timeout in milliseconds for the read-lock, if supported by the read-lock. If the read-lock could not be granted and the timeout triggered, then Camel will skip the file. At next poll Camel, will try the file again, and this time maybe the read-lock could be granted. Use a value of 0 or lower to indicate forever. Currently `fileLock`, `changed` and `rename` support the timeout.
`readLockCheck-Interval`	1000	Interval in milliseconds for the read-lock, if supported by the read lock. This interval is used for sleeping between attempts to acquire the read lock. For example when using the `changed` read lock, you can set a higher interval period to cater for *slow writes* . The default of 1 sec. may be *too fast* if the producer is very slow writing the file.
`directoryMust-Exist`	`false`	Similar to `startingDirectoryMustExist` but this applies during polling recursive sub directories.
`doneFileName`	`null`	If provided, Camel will only consume files if a *done* file exists. This option configures what file name to use. Either you can specify a fixed name, or you can use dynamic placeholders. The *done* file is **always** expected in the same folder as the original file. See *using done file* and *writing done file* sections for examples.
`exclusiveRead-LockStrategy`	`null`	Pluggable read-lock as a `org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy` implementation.
`maxMessages-PerPoll`	0	An integer that defines the maximum number of messages to gather per poll. By default, no maximum is set. It can be used to set a limit of, for example, 1000 to avoid having the server read thousands of files as it starts up. Set a value of 0 or negative to disable it. You can use the `eagerMaxMessagesPerPoll` option and set this to `false` to allow to scan all files first and then sort afterwards.
`eagerMaxMessagesPerPoll`	`true`	Allows for controlling whether the limit from `maxMessagesPerPoll` is eager or not. If eager then the limit is during the scanning of files. Whereas `false` would scan all files, and then perform sorting. Setting this option to `false` allows to sort all files first, and then limit the poll. Note that this requires a higher memory usage as all file details are in memory to perform the sorting.

Name	Default Value	Description
minDepth	0	The minimum depth to start processing when recursively processing a directory. Using minDepth=1 means the base directory. Using minDepth=2 means the first sub directory.
maxDepth	Integer.MAX_VALUE	The maximum depth to traverse when recursively processing a directory.
processStrategy	null	A pluggable org.apache.camel.component.file.GenericFileProcessStrategy allowing you to implement your own readLock option or similar. Can also be used when special conditions must be met before a file can be consumed, such as a special *ready* file exists. If this option is set then the readLock option does not apply.
startingDirectoryMustExist	false	whether the starting directory must exist. Keep in mind that the autoCreate option is default enabled, which means the starting directory is normally auto created if it doesn't exist. You can disable autoCreate and enable this to ensure the starting directory must exist. It will then throw an exception if the directory doesn't exist.
pollStrategy	null	A pluggable org.apache.camel.PollingConsumerPollStrategy allowing you to provide your custom implementation to control error handling usually occurred during the poll operation *before* an Exchange has been created and routed in Camel. In other words the error occurred while the polling was gathering information, for instance access to a file network failed so Camel cannot access it to scan for files. The default implementation will log the caused exception at WARN level and ignore it.
sendEmptyMessageWhenIdle	false	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.
consumer.bridgeErrorHandler	false	Allows for bridging the consumer to the Camel routing Error Handler, which mean any exceptions occurred while trying to pickup files, or the likes, will now be processed as a message and handled by the routing Error Handler. By default the consumer will use the org.apache.camel.spi.ExceptionHandler to deal with exceptions, that by default will be logged at WARN/ERROR level and ignored.

3.14.2.3. Default behavior for file consumer

- By default the file is locked for the duration of the processing.

- After the route has completed, files are moved into the .camel subdirectory, so that they appear to be deleted.

- The File Consumer will always skip any file whose name starts with a dot, such as ., .camel, .m2 or .groovy.

- Only files (not directories) are matched for valid filename, if options such as: include or exclude are used.

3.14.2.4. Producer

Name	Default Value	Description
fileExist	Override	this specifies what to do if a file already exists with the same name. The following values can be specified: **Override**, **Append**, **Fail** and **Ignore**.

Name	Default Value	Description
		• `Override`, which is the default, replaces the existing file. • `Append` adds content to the existing file. • `Fail` throws a `GenericFileOperation-Exception`, indicating that there is already an existing file. • `Ignore` silently ignores the problem and **does not** override the existing file, but assumes everything is okay.
`tempPrefix`	`null`	This option is used to write the file using a temporary name and then, after the write is complete, rename it to the real name. Can be used to identify files being written and also avoid consumers (not using exclusive read locks) reading in-progress files. Is often used by FTP when uploading big files.
`tempFileName`	`null`	The **same** as `tempPrefix` option but offering a more fine grained control on the naming of the temporary filename as it uses the File Language .
`keepLastModified`	`false`	If enabled, will keep the last modified timestamp from the source file (if any). This will use the `Exchange.FILE_LAST_MODIFIED` header to located the timestamp. This header can contain either a `java.util.Date` or `long` with the timestamp. If the timestamp exists and the option is enabled it will set this timestamp on the written file. **Note:** This option only applies to the **file** producer. You *cannot* use this option with any of the ftp producers.
`eagerDeleteTarget-File`	`true`	Whether or not to *eagerly delete* any existing target file. (This option only applies when you use `fileExists=Override` and the `tempFileName` option). You can use this to disable deleting the target file before the temp file is written. For example you may have large files and want the target file to persist while the temp file is being written. Setting `eagerDeleteTargetFile` to false ensures the target file is only deleted until the very last moment, just before the temp file is being renamed to the target filename.
`doneFileName`	`null`	**Camel 2.6:** If provided, then Camel will write a second *done* file when the original file has been written. The *done* file will be empty. This option configures what file name to use. Either you can specify a fixed name. Or you can use dynamic placeholders. The *done* file will **always** be written in the same folder as the original file. See *writing done file* section for examples.

3.14.2.5. Default behavior for file producer

By default it will override any existing file, if one exist with the same name.

Override is default

> `Override` is the default for the file producer. This is also the default file operation using `java.io.File` - and also the default for the FTP library we use in the camel-ftp component.

3.14.3. Move and Delete operations

Any move or delete operation is executed after the routing has completed (post command); so during processing of the `Exchange` the file is still located in the inbox folder.

Let's illustrate this with an example:

```
from("file://inbox?move=.done").to("bean:handleOrder");
```

When a file is dropped in the `inbox` folder, the file consumer notices this and creates a new `FileExchange` that is routed to the `handleOrder` bean. The bean then processes the `File` object. At this point in time the file is still located in the `inbox` folder. After the bean completes, and thus the route is completed, the file consumer will perform the move operation and move the file to the `.done` sub-folder.

The **move** and **preMove** options should be a directory name, which can be either relative or absolute. If relative, the directory is created as a sub-folder from within the folder where the file was consumed.

By default, Camel will move consumed files to the `.camel` sub-folder relative to the directory where the file was consumed.

If you want to delete the file after processing, the route should be:

```
from("file://inobox?delete=true").to("bean:handleOrder");
```

We have introduced a **pre** move operation to move files **before** they are processed. This allows you to mark which files have been scanned as they are moved to this sub folder before being processed.

```
from("file://inbox?preMove=inprogress").to("bean:handleOrder");
```

You can combine the **pre** move and the regular move:

```
from("file://inbox?preMove=inprogress&move=.done").to("bean:handleOrder");
```

So in this situation, the file is in the `inprogress` folder when being processed and after it is processed, it is moved to the `.done` folder.

3.14.3.1. Fine grained control over Move and PreMove option

The **move** and **preMove** option is Expression -based, so we have the full power of the File Language to do advanced configuration of the directory and name pattern. Camel will, in fact, internally convert the directory name you enter into a File Language expression. So, for example, when we enter `move=.done` Camel will convert this into: `${file:parent}/.done/${file:onlyname}`. This only happens if Camel detects that you have not provided a `${ }` in the option value. So when you enter a `${ }` Camel will **not** convert it and thus you have full control.

So, if we want to move the file into a backup folder with today's date as the pattern, we can do:

```
move=backup/${date:now:yyyyMMdd}/${file:name}
```

3.14.3.2. About moveFailed

The `moveFailed` option allows you to move files that **could not** be processed succesfully to another location such as a error folder of your choice. For example to move the files in an error folder with a timestamp you can use `moveFailed=/error/${file:name.noext}-${date:now:yyyyMMddHHmmssSSS}.${file:ext}`.

See more examples in File Language

3.14.4. Message Headers

The following headers are supported by this component:

3.14.4.1. File producer only

Header	Description
CamelFileName	Specifies the name of the file to write (relative to the endpoint directory). The name can be a `String` ; a `String` with a File Language or Simple expression; or an Expression object. If it is `null` then Camel will auto-generate a filename based on the message unique ID.
CamelFileNameProduced	The absolute filepath (path + name) for the output file that was written. This header is set by Camel and its purpose is providing end-users with the name of the file that was written.

3.14.4.2. File consumer only

Header	Description
CamelFileName	Name of the consumed file as a relative file path with offset from the starting directory configured on the endpoint.
CamelFileNameOnly	Just the file name (the name with no leading paths).
CamelFileAbsolute	A `boolean` option specifying whether the consumed file denotes an absolute path or not. It should normally be `false` for relative paths. Absolute paths should normally not be used but we added to the move option to allow moving files to absolute paths; it can also be used elsewhere.
CamelFileAbsolutePath	The absolute path to the file. For relative files this path holds the relative path instead.
CamelFilePath	The file path. For relative files this is the starting directory + the relative filename. For absolute files this is the absolute path.
CamelFileRelativePath	The relative path.
CamelFileParent	The parent path.
CamelFileLength	A `long` value containing the file size.
CamelFileLastModified	A `Date` value containing the last modified timestamp of the file.

3.14.5. Batch Consumer

This component implements the Batch Consumer .

3.14.5.1. Exchange Properties, file consumer only

As the file consumer is `BatchConsumer` it supports batching the files it polls. By batching it means that Camel will add some properties to the Exchange so you know the number of files polled, and the current index, in that order.

Property	Description
`CamelBatchSize`	The total number of files that was polled in this batch.
`CamelBatchIndex`	The current index of the batch. Starts from 0.
`CamelBatchComplete`	A `boolean` value indicating the last Exchange in the batch. Is only `true` for the last entry.

This would allow you, for example, to know how many files exist in the batch and use that information to let the Section 2.2, "Aggregator" aggregate that precise number of files.

3.14.6. Common gotchas with folder and filenames

When Camel is producing files (writing files) there are a few gotchas affecting how to set a filename of your choice. By default, Camel will use the message ID as the filename, and since the message ID is normally a unique generated ID, you will end up with filenames such as: `ID-MACHINENAME-2443-1211718892437-1-0`. If such a filename is not desired, then you must provide a filename in the `CamelFileName` message header. The constant, `Exchange.FILE_NAME`, can also be used.

The sample code below produces files using the message ID as the filename:

```
from("direct:report").to("file:target/reports");
```

To use `report.txt` as the filename you have to do:

```
from("direct:report").setHeader(Exchange.FILE_NAME, constant("report.txt"))
      .to( "file:target/reports");
```

... the same as above, but with `CamelFileName`:

```
from("direct:report").setHeader("CamelFileName", constant("report.txt"))
      .to( "file:target/reports");
```

An example of a syntax where we set the filename on the endpoint with the **fileName** URI option:

```
from("direct:report").to("file:target/reports/?fileName=report.txt");
```

3.14.7. Filename Expression

Filename can be set either using the **expression** option or as a string-based File Language expression in the `CamelFileName` header. See the File Language for syntax and samples.

3.14.8. Consuming files from folders where others drop files directly

Warning: there may be difficulties if you consume files from a directory where other applications directly write files. Please look at the different `readLock` options to see if they can help.

If you are writing files to the folder, then the best approach is to write to another folder and after the write, move the file in the drop folder.

However if you need to write files directly to the drop folder then the option `changed` could better detect whether a file is currently being written/copied. `changed` uses a file changed algorithm to see whether the file size or modification changes over a period of time. The other `readLock` options rely on Java File API which is not always good at detecting file changes. You may also want to look at the `doneFileName` option, which uses a marker file (done) to signal when a file is done and ready to be consumed.

3.14.9. Using done files

Available as of Camel 2.6

See also section *writing done files* below.

If you want only to consume files when a done file exists, then you can use the `doneFileName` option on the endpoint.

```
from("file:bar?doneFileName=done");
```

This will only consume files from the bar folder, if a file name done exists in the same directory as the target files. Camel will automatic delete the done file when it is finished consuming the files.

However it's more common to have one done file per target file. This means there is a 1:1 correlation. To do this you must use dynamic placeholders in the `doneFileName` option. Currently Camel supports the following two dynamic tokens: `file:name` and `file:name.noext` which must be enclosed in ${ }. The consumer only supports the static part of the done file name as either prefix or suffix (not both).

```
from("file:bar?doneFileName=${file:name}.done");
```

In this example only files will be polled if there exists a done file with the name *file name* .done. For example

- `hello.txt` is the file to be consumed

- `hello.txt.done` is the associated done file

You can also use a prefix for the done file, such as:

```
from("file:bar?doneFileName=ready-${file:name}");
```

- `hello.txt` is the file to be consumed

- `ready-hello.txt` is the associated done file

3.14.10. Writing done files

Available as of Camel 2.6

After you have written a file you may want to write an additional *done* file as a kind of marker, to indicate to others that the file is finished and has been written. To do that you can use the `doneFileName` option on the file producer endpoint.

```
.to("file:bar?doneFileName=done");
```

This will simply create a file named `done` in the same directory as the target file.

However it's more common to have one done file per target file. This means there is a 1:1 correlation. To do this you must use dynamic placeholders in the `doneFileName` option. Currently Camel supports the following two dynamic tokens: `file:name` and `file:name.noext` which must be enclosed in ${ }.

```
.to("file:bar?doneFileName=done-${file:name}");
```

This will for example create a file named `done-foo.txt` if the target file was `foo.txt` in the same directory as the target file.

```
.to("file:bar?doneFileName=${file:name}.done");
```

This will for example create a file named `foo.txt.done` if the target file was `foo.txt` in the same directory as the target file.

```
.to("file:bar?doneFileName=${file:name.noext}.done");
```

This will for example create a file named `foo.done` if the target file was `foo.txt` in the same directory as the target file.

3.14.11. Samples

3.14.11.1. Read from a directory and write to another directory

```
from("file://inputdir/?delete=true").to("file://outputdir")
```

Listen on a directory and create a message for each file dropped there. Copy the contents to the `outputdir` and delete the file in the `inputdir` .

3.14.11.2. Reading recursively from a directory and writing to another

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir")
```

Listen on a directory and create a message for each file dropped there. Copy the contents to the `outputdir` and delete the file in the `inputdir`. This will scan recursively into sub-directories, and lay out the files in the same directory structure in the `outputdir` as the `inputdir`, including any sub-directories.

```
inputdir/foo.txt
inputdir/sub/bar.txt
```

This will result in the following output layout:

```
outputdir/foo.txt
outputdir/sub/bar.txt
```


Using flatten

If you want to store the files in the outputdir directory in the same directory, disregarding the source directory layout (for example to flatten out the path), you add the `flatten=true` option on the file producer side:

```
from("file://inputdir/?recursive=true&delete=true")
        .to("file://outputdir?flatten=true")
```

This will result in the following output layout:

```
outputdir/foo.txt
outputdir/bar.txt
```

3.14.11.3. Reading from a directory and the default move operation

Camel will by default move any processed file into a `.camel` subdirectory in the directory the file was consumed from.

```
from("file://inputdir/?recursive=true&delete=true").to("file://outputdir")
```

Affects the layout as follows:

before

```
inputdir/foo.txt
inputdir/sub/bar.txt
```

after

```
inputdir/.camel/foo.txt
inputdir/sub/.camel/bar.txt
outputdir/foo.txt
outputdir/sub/bar.txt
```

3.14.11.4. Read from a directory and process the message in java

```
from("file://inputdir/").process(new Processor() {
  public void process(Exchange exchange) throws Exception {
    Object body = exchange.getIn().getBody();
    // do some business logic with the input body
    ...
  }
});
```

The body will be a `File` object that points to the file that was just dropped into the `inputdir` directory.

3.14.11.5. Writing to files

Camel is of course also able to write files, that is, produce files. In the sample below we receive some reports on the SEDA queue that we process before the reports are written to a directory.

```
public void testToFile() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:result");
    mock.expectedMessageCount(1);
    mock.expectedFileExists("target/test-reports/report.txt");

    template.sendBody("direct:reports", "This is a great report");

    assertMockEndpointsSatisfied();
}

protected JndiRegistry createRegistry() throws Exception {
    // bind our processor in the registry with the given id
    JndiRegistry reg = super.createRegistry();
    reg.bind("processReport", new ProcessReport());
```

```
    return reg;
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // the reports from the seda queue are processed by our
            // processor before they are written to files in the
            // target/reports directory
            from("direct:reports").processRef("processReport")
                    .to("file://target/test-reports", "mock:result");
        }
    };
}

private class ProcessReport implements Processor {

    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        // do some business logic here
        ...
        // set the output to the file
        exchange.getOut().setBody(body);

        // set the output filename using java code logic, notice that this
        // is done by setting a special header property of the out exchange
        exchange.getOut().setHeader(Exchange.FILE_NAME, "report.txt");
    }

}
```

3.14.11.6. Write to subdirectory using Exchange.FILE_NAME

Using a single route, it is possible to write a file to any number of subdirectories. If you have a route setup as such:

```
<route>
   <from uri="bean:myBean"/>
   <to uri="file:/rootDirectory"/>
</route>
```

You can have myBean set the header Exchange.FILE_NAME to values such as:

```
Exchange.FILE_NAME = hello.txt => /rootDirectory/hello.txt
Exchange.FILE_NAME = foo/bye.txt => /rootDirectory/foo/bye.txt
```

This allows you to have a single route to write files to multiple destinations.

3.14.11.7. Using expression for filenames

In this sample we want to move consumed files to a backup folder using today's date as a sub-folder name:

```
from("file://inbox?move=backup/${date:now:yyyyMMdd}/${file:name}").to("...");
```

See File Language for more samples.

3.14.12. Avoiding reading the same file more than once (idempotent consumer)

Camel supports Section 2.18, "Idempotent Consumer" directly within the component so it will skip already processed files. This feature can be enabled by setting the `idempotent=true` option.

```
from("file://inbox?idempotent=true").to("...");
```

By default Camel uses a in memory based store for keeping track of consumed files, it uses a least recently used cache holding up to 1000 entries. You can plugin your own implementation of this store by using the `idempotentRepository` option using the # sign in the value to indicate it is a referring to a bean in the Registry with the specified `id` .

```
<!-- define our store as a plain Spring bean -->
   <bean id="myStore" class="com.mycompany.MyIdempotentStore"/>

<route>
   <from uri=
      "file://inbox?idempotent=true&amp;idempotentRepository=#myStore"/>
   <to uri="bean:processInbox"/>
</route>
```

Camel will log at `DEBUG` level if it skips a file because it has been consumed before:

```
DEBUG FileConsumer is idempotent and the file has been consumed before.
This will skip this file: target\idempotent\report.txt
```

3.14.13. Filter using org.apache.camel.component.file.GenericFileFilter

Camel supports pluggable filtering strategies. You can then configure the endpoint with such a filter to skip certain files being processed.

In the sample we have built our own filter that skips files starting with `skip` in the filename:

```
public class MyFileFilter implements GenericFileFilter {
    public boolean accept(GenericFile pathname) {
        // we don't accept any files starting with skip in the name
        return !pathname.getFileName().startsWith("skip");
    }
}
```

Then we can configure our route using the **filter** attribute to reference our filter (using # notation) that we have defines in the Spring XML file:

```
<!-- define our sorter as a plain Spring bean -->
   <bean id="myFilter" class="com.mycompany.MyFileSorter"/>

  <route>
    <from uri="file://inbox?filter=#myFilter"/>
    <to uri="bean:processInbox"/>
  </route>
```

3.14.13.1. Filtering using ANT path matcher

> **Tip**
>
> There are also `antInclude` and `antExclude` options to make it easy to specify ANT style include/exclude without having to define the filter. See the URI options above for more information.

The ANT path matcher is shipped out-of-the-box in the **camel-spring** jar. So you need to depend on **camel-spring** if you are using Maven. The reason is that we leverage Spring's AntPathMatcher to do the matching.

The file paths is matched with the following rules:

- ? matches one character

- * matches zero or more characters

- ** matches zero or more directories in a path

The sample below demonstrates how to use it:

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

    <!-- use myFilter as filter to allow setting
            ANT paths for which files to scan for -->
    <endpoint id="myFileEndpoint" uri=
        "file://target/antpathmatcher?recursive=true&filter=#myAntFilter"/>

    <route>
        <from ref="myFileEndpoint"/>
        <to uri="mock:result"/>
    </route>
</camelContext>

<!-- we use the antpath file filter to use ant paths -->
<!-- for includes and excludes -->
<bean id="myAntFilter"
    class="org.apache.camel.component.file.AntPathMatcherGenericFileFilter">
    <!-- include and file in the subfolder that has 'day' in the name -->
    <property name="includes" value="**/subfolder/**/*day*"/>
    <!-- exclude all files with 'bad' in name or .xml files. -->
    <!-- Use comma to separate multiple excludes -->
    <property name="excludes" value="**/*bad*,**/*.xml"/>
</bean>
```

3.14.14. Sorting using Comparator

Camel supports pluggable sorting strategies. This strategy it to use the built in `java.util.Comparator` in Java. You can then configure the endpoint with such a comparator and have Camel sort the files before being processed.

In the sample we have built our own comparator that sorts by file name:

```
public class MyFileSorter implements Comparator<GenericFile> {
```

```
    public int compare(GenericFile o1, GenericFile o2) {
        return o1.getFileName().compareToIgnoreCase(o2.getFileName());
    }
}
```

Then we can configure our route using the **sorter** option to reference to our sorter (mySorter) we have defined in the Spring XML file:

```
<!-- define our sorter as a plain Spring bean -->
   <bean id="mySorter" class="com.mycompany.MyFileSorter"/>

  <route>
    <from uri="file://inbox?sorter=#mySorter"/>
    <to uri="bean:processInbox"/>
  </route>
```

URI options can reference beans using the # syntax

In the Spring DSL route, notice that we can refer to beans in the Registry by prefixing the id with # . So writing sorter=#mySorter, will instruct Camel to go look in the Registry for a bean with the ID, mySorter .

3.14.15. Sorting using sortBy

Camel supports pluggable sorting strategies. This strategy it to use the File Language to configure the sorting. The sortBy option is configured as follows:

```
sortBy=group 1;group 2;group 3;...
```

where each group is separated with semi colon. In the simple situations you just use one group, so a simple example could be:

```
sortBy=file:name
```

This will sort by file name, you can reverse the order by prefixing reverse: to the group, so the sorting is now Z..A:

```
sortBy=reverse:file:name
```

As we have the full power of File Language we can use some of the other parameters, so if we want to sort by file size we do:

```
sortBy=file:length
```

You can configure to ignore the case, using ignoreCase: for string comparison, so if you want to use file name sorting but to ignore the case then we do:

```
sortBy=ignoreCase:file:name
```

You can combine ignore case and reverse, however reverse must be specified first:

```
sortBy=reverse:ignoreCase:file:name
```

In the sample below we want to sort by last modified file, so we do:

```
sortBy=file:modifed
```

Then we want to group by name as a second option so files with same modification is sorted by name:

```
sortBy=file:modifed;file:name
```

Now there is an issue here, can you spot it? Well the modified timestamp of the file is too fine as it will be in milliseconds, but what if we want to sort by date only and then subgroup by name? Well as we have the true power of File Language we can use the its date command that supports patterns. So this can be solved as:

```
sortBy=date:file:yyyyMMdd;file:name
```

That is powerful. You can also use reverse per group, so we could reverse the file names:

```
sortBy=date:file:yyyyMMdd;reverse:file:name
```

3.14.16. Using GenericFileProcessStrategy

The option `processStrategy` can be used to use a custom `GenericFileProcessStrategy` that allows you to implement your own *begin*, *commit* and *rollback* logic. For instance let's assume a system writes a file in a folder you should consume. But you should not start consuming the file before another *ready* file have been written as well.

So by implementing our own `GenericFileProcessStrategy` we can implement this as:

- In the `begin()` method we can test whether the special *ready* file exists. The begin method returns a `boolean` to indicate if we can consume the file or not.

- in the `commit()` method we can move the file and also delete the *ready* file.

3.15. Flatpack

3.15.1. Flatpack Component

The Flatpack component supports fixed width and delimited file parsing via the FlatPack library.

> **Notice:**
>
> This component only supports consuming from flatpack files to Object model. You can not (yet) write from Object model to flatpack format.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-flatpack</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.15.1.1. URI format

```
flatpack:[delim|fixed]:flatPackConfig.pzmap.xml[?options]
```

Or for a delimited file handler with no configuration file use

```
flatpack:someName[?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.15.1.2. URI Options

Name	Default Value	Description
`delimiter`	`,`	The default character delimiter for delimited files.
`textQualifier`	`"`	The text qualifier for delimited files.
`ignoreFirstRecord`	`true`	Whether the first line is ignored for delimited files (for the column headers).
`splitRows`	`true`	The component can either process each row one by one or the entire content at once.

3.15.1.3. Examples

- `flatpack:fixed:foo.pzmap.xml` creates a fixed-width endpoint using the `foo.pzmap.xml` file configuration.

- `flatpack:delim:bar.pzmap.xml` creates a delimited endpoint using the `bar.pzmap.xml` file configuration.

- `flatpack:foo` creates a delimited endpoint called `foo` with no file configuration.

3.15.1.4. Message Headers

Camel will store the following headers on the IN message:

Header	Description
`camelFlatpackCounter`	The current row index. For `splitRows=false` the counter is the total number of rows.

3.15.1.5. Message Body

The component delivers the data in the IN message as a `org.apache.camel.component.flatpack.DataSetList` object that has converters for

- `java.util.Map`

- `java.util.List`.

Usually you want the `Map` if you process one row at a time (`splitRows=true`). Use `List` for the entire content (`splitRows=false`), where each element in the list is a `Map` . Each `Map` contains the key for the column name and its corresponding value.

For example to get the firstname from the sample below:

```
Map row = exchange.getIn().getBody(Map.class);
  String firstName = row.get("FIRSTNAME");
```

However, you can also always get it as a `List` (even for `splitRows=true`). The same example:

```
List data = exchange.getIn().getBody(List.class);
  Map row = (Map)data.get(0);
  String firstName = row.get("FIRSTNAME");
```

3.15.1.6. Header and Trailer records

The header and trailer notions in Flatpack are supported. However, you **must** use fixed record IDs:

• `header` for the header record (must be lowercase)

• `trailer` for the trailer record (must be lowercase)

The example below illustrates this fact that we have a header and a trailer. You can omit one or both of them if not needed.

```
<RECORD id="header" startPosition="1" endPosition="3" indicator="HBT">
        <COLUMN name="INDICATOR" length="3"/>
        <COLUMN name="DATE" length="8"/>
    </RECORD>

    <COLUMN name="FIRSTNAME" length="35" />
    <COLUMN name="LASTNAME" length="35" />
    <COLUMN name="ADDRESS" length="100" />
    <COLUMN name="CITY" length="100" />
    <COLUMN name="STATE" length="2" />
    <COLUMN name="ZIP" length="5" />

    <RECORD id="trailer" startPosition="1" endPosition="3" indicator="FBT">
        <COLUMN name="INDICATOR" length="3"/>
        <COLUMN name="STATUS" length="7"/>
    </RECORD>
```

3.15.1.7. Using the endpoint

A common use case is sending a file to this endpoint for further processing in a separate route. For example:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
    <route>
        <from uri="file://someDirectory"/>
        <to uri="flatpack:foo"/>
    </route>

    <route>
        <from uri="flatpack:foo"/>
        ...
    </route>
</camelContext>
```

You can also convert the payload of each message created to a `Map` for easy Bean Integration

3.15.2. Flatpack DataFormat

The Section 3.15, "Flatpack" component ships with the Flatpack data format that can be used to format between fixed width or delimited text messages to a `List` of rows as `Map` .

• marshal = from `List<Map<String, Object>>` to `OutputStream` (can be converted to `String`)

• unmarshal = from `java.io.InputStream` (such as a `File` or `String`) to a `java.util.List` as an `org.apache.camel.component.flatpack.DataSetList` instance. The result of the operation will

contain all the data. If you need to process each row one by one you can split the exchange, using Section 2.48, "Splitter" .

Notice: The Flatpack library does currently not support header and trailers for the marshal operation.

3.15.2.1. Options

The data format has the following options:

Option	Default	Description
definition	null	The flatpack pzmap configuration file. Can be omitted in simpler situations, but it is preferred to use the pzmap.
fixed	false	Delimited or fixed.
ignoreFirstRecord	true	Whether the first line is ignored for delimited files (for the column headers).
textQualifier	"	If the text is qualified with a char such as " .
delimiter	,	The delimiter char (could be ; , or similar)
parserFactory	null	Uses the default Flatpack parser factory.

3.15.2.2. Usage

To use the data format, simply instantiate an instance and invoke the marhsal or unmarshal operation in the route builder:

```
FlatpackDataFormat fp = new FlatpackDataFormat();
  fp.setDefinition(new ClassPathResource("INVENTORY-Delimited.pzmap.xml"));
  ...
  from("file:order/in").unmarshal(df).to("seda:queue:neworder");
```

The sample above will read files from the `order/in` folder and unmarshal the input using the Flatpack configuration file `INVENTORY-Delimited.pzmap.xml` that configures the structure of the files. The result is a `DataSetList` object we store on the SEDA queue.

```
FlatpackDataFormat df = new FlatpackDataFormat();
df.setDefinition(new ClassPathResource("PEOPLE-FixedLength.pzmap.xml"));
df.setFixed(true);
df.setIgnoreFirstRecord(false);

from("seda:people").marshal(df).convertBodyTo(String.class)
      .to("jms:queue:people");
```

In the code above we marshal the data from a Object representation as a `List` of rows as `Maps`. The rows as `Map` contains the column name as the key, and the corresponding value. This structure can be created in Java code (for example from a processor). We marshal the data according to the Flatpack format and convert the result as a `String` object and store it on a JMS queue.

3.15.2.3. Dependencies

To use Flatpack in your Camel routes, you need to add the a dependency on **camel-flatpack** which implements this data format.

If you use Maven you could add the following to your pom.xml, substituting the version number for the latest release (see the download page for the latest versions).

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-flatpack</artifactId>
    <version>1.5.0</version>
</dependency>
```

3.16. Freemarker

The freemarker component allows for processing a message using a FreeMarker template. This can be ideal when using Templating to generate responses for requests.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-freemarker</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.16.1. URI format

```
freemarker:templateName[?options]
```

where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (for example: file://folder/myfile.ftl").

You can append query options to the URI in the following format, ?option=value&option=value&...

3.16.2. Options

Option	Default	Description
contentCache	true	Cache for the resource content when it is loaded. Note: Cached resource content can be cleared via JMX using the endpoint's {{clearContentCache}} operation.
encoding	null	Character encoding of the resource content.
templateUpdateDelay	5	Character encoding of the resource content.

3.16.3. Headers

Headers set during the FreeMarker evaluation are returned to the message and added as headers. This provides a mechanism for the FreeMarker component to return values to the Message.

An example: Set the header value of fruit in the Freemarker template:

```
${request.setHeader('fruit', 'Apple')}
```

The header, `fruit`, is now accessible from the `message.out.headers` .

3.16.4. Freemarker Context

Camel will provide exchange information in the Freemarker context (just a `Map`). The `Exchange` is transferred as:

key	value
`exchange`	The Exchange itself.
`exchange.properties`	The Exchange properties.
`headers`	The headers of the In message.
`camelContext`	The Camel Context.
`request`	The In message.
`body`	The In message body.
`response`	The Out message (only for InOut message exchange pattern).

3.16.5. Hot reloading

The Freemarker template resource is by default **not** hot reloadable for both file and classpath resources (expanded jar). If you set `contentCache=false`, then Camel will not cache the resource and hot reloading is thus enabled. This scenario can be used in development.

3.16.6. Dynamic templates

Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

Header	Type	Description
FreemarkerConstants. FREEMARKER_RESOURCE_URI	String	A URI for the template resource to use instead of the endpoint configured.
FreemarkerConstants. FREEMARKER_TEMPLATE	String	The template to use instead of the endpoint configured.

3.16.7. Samples

For example you could use something like:

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

to use a FreeMarker template to formulate a response for a message for InOut message exchanges (where there is a `JMSReplyTo` header).

If you want to use InOnly and consume the message and send it to another destination you could use:

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

To disable the content cache, for example, for development usage where the `.ftl` template should be hot reloaded:

```
from("activemq:My.Queue")
  .to("freemarker:com/acme/MyResponse.ftl?contentCache=false")
  .to("activemq:Another.Queue");
```

A file-based resource:

```
from("activemq:My.Queue")
  .to("freemarker:file://myfolder/MyResponse.ftl?contentCache=false")
  .to("activemq:Another.Queue");
```

In it is possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").setHeader(FreemarkerConstants.FREEMARKER_RESOURCE_URI).
          constant("path/to/my/template.ftl").to("freemarker:dummy");
```

3.17. FTP

This component provides access to remote file systems over the FTP and SFTP protocols.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-ftp</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

⚠ Using FTPS

FTPS (also known as FTP Secure) is an extension to FTP that adds support for the Transport Layer Security (TLS) and the Secure Sockets Layer (SSL) cryptographic protocols.

⚠ Libraries used

This component uses two different libraries for the FTP work. FTP and FTPS uses Apache Commons Net while SFTP uses JCraft JSCH .

3.17.1. URI format

```
ftp://[username@]hostname[:port]/directoryname[?options]
sftp://[username@]hostname[:port]/directoryname[?options]
ftps://[username@]hostname[:port]/directoryname[?options]
```

where **directoryname** represents the underlying directory, which can contain nested folders.

If no **username** is provided, then `anonymous` login is attempted using no password. If no **port** number is provided, Camel will provide default values according to the protocol (ftp = 21, sftp = 22, ftps = 2222).

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.17.2. URI Options

The options below are exclusive for the Section 3.17, "FTP" component.

Name	Default Value	Description
username	null	Specifies the username to use to log into the remote file system.
password	null	Specifies the password to use to log into the remote file system.
binary	false	Specifies the file transfer mode, BINARY or ASCII. Default is ASCII (`false`).
disconnect	false	Whether or not to disconnect from remote FTP server right after use. Can be used for both consumer and producer. Disconnect will only disconnect the current connection to the FTP server. If you have a consumer which you want to stop, then you need to stop the consumer/route instead.
localWorkDirectory	null	When consuming, a local work directory can be used to store the remote file content directly in local files, to avoid loading the content into memory. This is beneficial if you consume a very big remote file and thus can conserve memory. See below for more details.
passiveMode	false	**FTP and FTPS only** : Specifies whether to use passive mode connections. Default is active mode (`false`).
securityProtocol	TLS	**FTPS only:** Sets the underlying security protocol. The following values are defined: `TLS` : Transport Layer Security `SSL` : Secure Sockets Layer
disableSecureDataChannelDefaults	false	**FTPS only** : Whether or not to disable using default values for `execPbsz` and `execProt` when using secure data transfer. You can set this option to `true` if you want to be in absolute full control what the options `execPbsz` and `execProt` should be used.
execProt	null	**FTPS only** : This will use option `P` by default, if secure data channel defaults hasn't been disabled. Possible values are: `C` : Clear `S` : Safe (SSL protocol only) `E` : Confidential (SSL protocol only) `P` : Private
execPbsz	null	**FTPS only** : This option specifies the buffer size of the secure data channel. If option `useSecureDataChannel` has been enabled and this option has not been explicit set, then value `0` is used.
isImplicit	false	**FTPS only:** Sets the security mode(implicit/explicit). Default is explicit (`false`).
knownHostsFile	null	**SFTP only:** Sets the `known_hosts` file, so that the SFTP endpoint can do host key verification.
privateKeyFile	null	**SFTP only:** Set the private key file to that the SFTP endpoint can do private key verification.

Name	Default Value	Description
privateKeyFilePassphrase	null	**SFTP only:** Set the private key file passphrase to that the SFTP endpoint can do private key verification.
strictHostKeyChecking	no	**SFTP only:** Sets whether to use strict host key checking. Possible values are: no, yes and ask. **Note:** ask does not make sense to use as Camel cannot answer the question for you as it is meant for human intervention.
maximumReconnectAttempts	3	Specifies the maximum reconnect attempts Camel performs when it tries to connect to the remote FTP server. Use 0 to disable this behavior.
reconnectDelay	1000	Delay in milliseconds Camel will wait before performing a reconnect attempt.
connectTimeout	10000	the connect timeout in milliseconds. This corresponds to using ftpClient.connectTimeout for the FTP/FTPS. For SFTP this option is also used when attempting to connect.
soTimeout	null	**FTP and FTPS Only:** the SocketOptions.SO_TIMEOUT value in milliseconds. Note SFTP will automatic use the connectTimeout as the soTimeout.
timeout	30000	**FTP and FTPS Only:** the data timeout in milliseconds. This corresponds to using ftpClient.dataTimeout for the FTP/FTPS. For SFTP there is no data timeout.
throwExceptionOnConnect-Failed	false	Whether or not to throw an exception if a successful connection and login could not be established. This allows a custom pollStrategy to deal with the exception, for example to stop the consumer.
siteCommand	null	**FTP and FTPS Only:** To execute site commands after successful login. Multiple site commands can be separated using a new line character (\n). Use help site to see which site commands your FTP server supports.
stepwise	true	**Camel 2.6 onwards:** Whether or not stepwise traversing directories should be used or not. Stepwise means that it will 'cd' one directory at a time. See more details below. You can disable this in case you can't use this approach.
separator	Auto	**Camel 2.6 onwards:** Dictates what path separator char to use when uploading files. Auto = Use the path provided without altering it. UNIX = Use unix style path separators. Windows = Use Windows style path separators.
ftpClient	null	**FTP and FTPS Only:** Allows you to use a custom org.apache.commons.net.ftp.FTPClient instance.
ftpClientConfig	null	**FTP and FTPS Only:** Allows you to use a custom org.apache.commons.net.ftp.FTPClientConfig instance.
ftpClient.trustStore.file	null	**FTPS Only:** Sets the trust store file, so that the FTPS client can look up for trusted certificates.
ftpClient.trustStore.type	JKS	**FTPS Only:** Sets the trust store type.
ftpClient.trustStore.algorithm	SunX509	**FTPS Only:** Sets the trust store algorithm.

Name	Default Value	Description
`ftpClient.trustStore. password`	`null`	**FTPS Only:** Sets the trust store password.
`ftpClient.keyStore.file`	`null`	**FTPS Only:** Sets the key store file, so that the FTPS client can look up for the private certificate.
`ftpClient.keyStore.type`	`JKS`	**FTPS Only:** Sets the key store type.
`ftpClient.keyStore. algorithm`	`SunX509`	**FTPS Only:** Sets the key store algorithm.
`ftpClient.keyStore. password`	`null`	**FTPS Only:** Sets the key store password.
`ftpClient.keyStore. keyPassword`	`null`	**FTPS Only:** Sets the private key password.

FTPS component default trust store

By default, the FTPS component trust store accepts all certificates. If you only want to trust selective certificates, you have to configure the trust store with the `ftpClient.trustStore.xxx` options or by configuring a custom `ftpClient`.

You can configure additional options on the `ftpClient` and `ftpClientConfig` from the URI directly by using the `ftpClient.` or `ftpClientConfig.` prefix.

For example to set the `setDataTimeout` on the `FTPClient` to 30 seconds you can do:

```
from("ftp://foo@myserver?password=secret&ftpClient.dataTimeout=30000")
    .to("bean:foo");
```

You can mix and match and have use both prefixes, for example to configure date format or timezones.

```
from("ftp://foo@myserver?password=secret&ftpClient.dataTimeout=30000&" +
    "ftpClientConfig.serverLanguageCode=fr").to("bean:foo");
```

You can have as many of these options as you like.

See the documentation of the Apache Commons FTP FTPClientConfig for possible options and more details, and also Apache Commons FTP FTPClient.

If you do not like having complex configurations inserted in the url you can use `ftpClient` or `ftpClientConfig` by letting Camel look in the Registry for it. For example:

```
<bean id="myConfig" class="org.apache.commons.net.ftp.FTPClientConfig">
    <property name="lenientFutureDates" value="true"/>
    <property name="serverLanguageCode" value="fr"/>
</bean>
```

And then let Camel lookup this bean when you use the # notation in the url.

```
from("ftp://foo@myserver?password=secret&ftpClientConfig=#myConfig")
        .to("bean:foo");
```

3.17.3. More URI options

⚠️ **Important**

See Section 3.14, "File" as all the options there also apply to this component.

3.17.4. Stepwise changing directories

Camel FTP can operate in two modes in terms of traversing directories when consuming files (for example, downloading) or producing files (for example, uploading)

• stepwise

• not stepwise

You may want to pick either one (Camel 2.6 onwards) depending on your situation and security issues (some Camel end users can only download files if they use stepwise, while others can only download if they do not). You can use the stepwise option to control the behavior.

Note that stepwise changing of directory will in most cases only work when the user is confined to its home directory and when the home directory is reported as "/" .

The difference between the two behaviours is best illustrated with an example. Suppose we have the following directory structure on the remote FTP server we need to traverse and download files:

```
/
/one
/one/two
/one/two/sub-a
/one/two/sub-b
```

And that we have a file in each of sub-a (a.txt) and sub-b (b.txt) folder.

3.17.4.1. Using stepwise=true (default mode)

```
TYPE A
200 Type set to A
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
SYST
215 UNIX emulated by FileZilla
PORT 127,0,0,1,17,94
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CWD sub-a
250 CWD successful. "/one/two/sub-a" is current directory.
PORT 127,0,0,1,17,95
200 Port command successful
```

```
LIST
150 Opening data channel for directory list.
226 Transfer OK
CDUP
200 CDUP successful. "/one/two" is current directory.
CWD sub-b
250 CWD successful. "/one/two/sub-b" is current directory.
PORT 127,0,0,1,17,96
200 Port command successful
LIST
150 Opening data channel for directory list.
226 Transfer OK
CDUP
200 CDUP successful. "/one/two" is current directory.
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
PORT 127,0,0,1,17,97
200 Port command successful
RETR foo.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
CWD sub-a
250 CWD successful. "/one/two/sub-a" is current directory.
PORT 127,0,0,1,17,98
200 Port command successful
RETR a.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
250 CWD successful. "/" is current directory.
PWD
257 "/" is current directory.
CWD one
250 CWD successful. "/one" is current directory.
CWD two
250 CWD successful. "/one/two" is current directory.
CWD sub-b
250 CWD successful. "/one/two/sub-b" is current directory.
PORT 127,0,0,1,17,99
200 Port command successful
RETR b.txt
150 Opening data channel for file transfer.
226 Transfer OK
CWD /
```

```
250 CWD successful. "/" is current directory.
QUIT
221 Goodbye
disconnected.
```

As you can see when stepwise is enabled, it will traverse the directory structure using CD xxx.

3.17.4.2. Using stepwise=false

```
230 Logged on
TYPE A
200 Type set to A
SYST
215 UNIX emulated by FileZilla
PORT 127,0,0,1,4,122
200 Port command successful
LIST one/two
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,123
200 Port command successful
LIST one/two/sub-a
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,124
200 Port command successful
LIST one/two/sub-b
150 Opening data channel for directory list
226 Transfer OK
PORT 127,0,0,1,4,125
200 Port command successful
RETR one/two/foo.txt
150 Opening data channel for file transfer.
226 Transfer OK
PORT 127,0,0,1,4,126
200 Port command successful
RETR one/two/sub-a/a.txt
150 Opening data channel for file transfer.
226 Transfer OK
PORT 127,0,0,1,4,127
200 Port command successful
RETR one/two/sub-b/b.txt
150 Opening data channel for file transfer.
226 Transfer OK
QUIT
221 Goodbye
disconnected.
```

As you can see when not using stepwise, there are no CD operation invoked at all.

3.17.5. Examples

```
ftp://someone@someftpserver.com/public/upload/images/holiday2008?password=
secret&binary=true
ftp://someoneelse@someotherftpserver.co.uk:12049/reports/2008/password=
secret&binary=false
ftp://publicftpserver.com/download
```

FTP Consumer does not support concurrency

The FTP consumer (with the same endpoint) does not support concurrency (the backing FTP client is not thread safe). You can use multiple FTP consumers to poll from different endpoints. It is only a single endpoint that does not support concurrent consumers.

The FTP producer does **not** have this issue, it supports concurrency.

In the future we will add consumer pooling to Camel to allow this consumer to support concurrency as well.

More information

This component is an extension of the Section 3.14, "File" component, and there are more samples and details on the Section 3.14, "File" component page.

3.17.6. Default when consuming files

The FTP consumer will by default leave the consumed files untouched on the remote FTP server. You have to configure it explicitly if you want it to delete the files or move them to another location. For example, you can use `delete=true` to delete the files, or use `move=.done` to move the files into a hidden done subdirectory.

The regular File consumer is different as it will (by default) move files to a `.camel` sub directory. The reason Camel does **not** do this by default for the FTP consumer is that it may lack permissions by default to be able to move or delete files.

3.17.6.1. limitations

The option **readLock** can be used to force Camel **not** to consume files that is currently in the progress of being written. However, this option is turned off by default, as it requires that the user has write access. There are only a few options supported for FTP. There are other solutions to avoid consuming files that are currently being written over FTP; for instance, you can write the file to a temporary destination and move the file after it has been written.

When moving files using `move` or `preMove` option the files are restricted to the FTP_ROOT folder. That prevents you from moving files outside the FTP area. If you want to move files to another area, you can use soft links and move files into a soft linked folder.

3.17.7. Message Headers

The following message headers can be used to affect the behavior of the component

Header	Description
`CamelFileName`	Specifies the output file name (relative to the endpoint directory) to be used for the output message when sending to the endpoint. If neither `CamelFileName` or an expression are specified, then a generated message ID is used as the filename instead.
`CamelFileNameProduced`	The absolute filepath (path + name) for the output file that was written. This header is set by Camel and its purpose is providing end-users the name of the file that was written.
`CamelFileBatchIndex`	Current index out of total number of files being consumed in this batch.
`CamelFileBatchSize`	Total number of files being consumed in this batch.
`CamelFileHost`	The remote hostname.
`CamelFileLocalWorkPath`	Path to the local work file, if local work directory is used.

3.17.8. About timeouts

The two set of libraries (see top) has different API for setting timeout. You can use the `connectTimeout` option for both of them to set a timeout in milliseconds to establish a network connection. An individual `soTimeout` can also be set on the FTP/FTPS, which corresponds to using `ftpClient.soTimeout`. Notice SFTP will automatically use `connectTimeout` as its `soTimeout`. The `timeout` option only applies for FTP/FTSP as the data timeout, which corresponds to the `ftpClient.dataTimeout` value. All timeout values are in milliseconds.

3.17.9. Using Local Work Directory

Camel supports consuming from remote FTP servers and downloading the files directly into a local work directory. This avoids reading the entire remote file content into memory as it is streamed directly into the local file using `FileOutputStream`.

Camel will store to a local file with the same name as the remote file, though with `.inprogress` as extension while the file is being downloaded. Afterwards, the file is renamed to remove the `.inprogress` suffix. And finally, when the Exchange is complete the local file is deleted.

So if you want to download files from a remote FTP server and store it as files then you need to route to a file endpoint such as:

```
from("ftp://someone@someserver.com?password=secret&localWorkDirectory=/tmp")
      .to("file://inbox");
```

Optimization by renaming work file

The route above is ultra efficient as it avoids reading the entire file content into memory. It will download the remote file directly to a local file stream. The `java.io.File` handle is then used as the Exchange body. The file producer leverages this fact and can work directly on the work file `java.io.File` handle and perform a `java.io.File.rename` to the target filename. As Camel knows it is a local work file, it can optimize and use a rename instead of a file copy, as the work file is meant to be deleted anyway.

3.17.10. Samples

In the sample below we set up Camel to download all the reports from the FTP server once every hour (60 min) as BINARY content and store it as files on the local file system.

```
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
    public void configure() throws Exception {
        // we use a delay of 60 minutes
        //(for example, once per hour we poll the FTP server)
        long delay = 60 * 60 * 1000L;

        // from the given FTP server we poll (= download) all the files
        // from the public/reports folder as BINARY types and store this as
        // files in a local directory. Camel will use the filenames from the

        // FTPServer. Notice that the FTPConsumer properties must be prefixed
        // with "consumer.". In the URL the delay parameter is from the
        // FileConsumer component so we should use consumer.delay as
        // the URI parameter name. The FTP Component is an extension of the
        // File Component.
        from("ftp://tiger:scott@localhost/public/reports?binary=true&
consumer.delay=" + delay).to("file://target/test-reports");
        }
    };
}
```

And the route using Spring DSL:

```
<route>
    <from uri="ftp://scott@localhost/public/reports?password=
tiger&amp;binary=true&amp;delay=60000"/>

    <to uri="file://target/test-reports"/>
</route>
```

3.17.10.1. Consuming a remote FTPS server (implicit SSL) and client authentication

```
from(
"ftps://admin@localhost:2222/public/camel?password=admin&securityProtocol=
SSL&isImplicit=true&ftpClient.keyStore.file=./src/test/resources/server.jks
&ftpClient.keyStore.password=password&
ftpClient.keyStore.keyPassword=password").to("bean:foo");
```

3.17.10.2. Consuming a remote FTPS server (explicit TLS) and a custom trust store configuration

```
from("ftps://admin@localhost:2222/public/camel?password=admin&ftpClient.
trustStore.file=./src/test/resources/server.jks&ftpClient.trustStore.
password=password").to("bean:foo");
```

3.17.11. Filter using org.apache.camel.component.file.GenericFileFilter

Camel supports pluggable filtering strategies. This strategy it to use the built in `org.apache.camel.component.file.GenericFileFilter` in Java. You can then configure the endpoint with such a filter to skip certain filters before being processed.

In the sample we have build our own filter that only accepts files starting with report in the filename.

```java
public class MyFileFilter implements GenericFileFilter {

    public boolean accept(GenericFile file) {
        // we only want report files
        return file.getFileName().startsWith("report");
    }
}
```

And then we can configure our route using the **filter** attribute to reference our filter (using # notation) that we have defined in the Spring XML file:

```xml
<!-- define our sorter as a plain Spring bean -->
   <bean id="myFilter" class="com.mycompany.MyFileFilter"/>

  <route>
    <from uri=
    "ftp://someuser@someftpserver.com?password=secret&amp;filter=#myFilter"/>
    <to uri="bean:processInbox"/>
  </route>
```

3.17.12. Filtering using ANT path matcher

The ANT path matcher is a filter that is shipped out-of-the-box in the **camel-spring** jar. So you need to depend on **camel-spring** if you are using Maven. The reason is that we leverage Spring's AntPathMatcher to do the matching.

The file paths are matched with the following rules:

- ? matches one character

- * matches zero or more characters

- ** matches zero or more directories in a path

The sample below demonstrates how to use it:

```xml
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <template id="camelTemplate"/>
    <!-- use myFilter as filter to allow setting ANT paths for which -->
    <!-- filesto scan for -->
    <endpoint id="myFTPEndpoint"
```

```
    uri="ftp://admin@localhost:20123/antpath?password=admin&recursive=true&
delay=10000&initialDelay=2000&filter=#myAntFilter"/>

    <route>
        <from ref="myFTPEndpoint"/>
        <to uri="mock:result"/>
    </route>
</camelContext>

<!-- we use the AntPathMatcherRemoteFileFilter to use ant paths for -->
<!-- includes and excludes -->
<bean id="myAntFilter"
   class="org.apache.camel.component.file.AntPathMatcherGenericFileFilter">

   <!-- include and file in the subfolder that has day in the name -->
   <property name="includes" value="**/subfolder/**/*day*"/>
   <!-- exclude all files with bad in name or .xml files. -->
   <!-- Use comma to separate multiple excludes -->
   <property name="excludes" value="**/*bad*,**/*.xml"/>
</bean>
```

3.17.13. Debug logging

This component has log level **TRACE** that can be helpful if you have problems.

3.18. Hibernate

The **hibernate:** component allows you to work with databases using Hibernate as the object relational mapping technology to map POJOs to database tables. The **camel-hibernate** library is provided by the Camel Extra project which hosts all *GPL related components for Camel.

3.18.1. Sending to the endpoint

Sending POJOs to the hibernate endpoint inserts entities into the database. The body of the message is assumed to be an entity bean that you have mapped to a relational table using the hibernate `.hbm.xml` files.

If the body does not contain an entity bean, use a Section 2.29, "Message Translator" in front of the endpoint to perform the necessary conversion first.

3.18.2. Consuming from the endpoint

Consuming messages removes (or updates) entities in the database. This allows you to use a database table as a logical queue; consumers take messages from the queue and then delete/update them to logically remove them from the queue.

If you do not wish to delete the entity when it has been processed, you can specify `consumeDelete=false` on the URI. This will result in the entity being processed each poll.

If you would rather perform some update on the entity to mark it as processed (such as to exclude it from a future query) then you can annotate a method with @Consumed which will be invoked on your entity bean when the entity bean is consumed.

3.18.3. URI format

```
hibernate:[entityClassName][?options]
```

For sending to the endpoint, the **entityClassName** is optional. If specified it is used to help use the Type Conversion to ensure the body is of the correct type.

For consuming the **entityClassName** is mandatory.

You can append query options to the URI in the following format, ?option=value&option=value&...

3.18.4. Options

Name	Default Value	Description
entityType	*entityClassName*	the provided *entityClassName* from the URI.
consumeDelete	true	Option for HibernateConsumer only. Specifies whether or not the entity is deleted after it is consumed.
consumeLockEntity	true	Option for HibernateConsumer only. Specifies whether or not to use exclusive locking of each entity while processing the results from the pooling.
flushOnSend	true	Option for HibernateProducer only. Flushes the EntityManager after the entity bean has been persisted.
maximumResults	-1	Option for HibernateConsumer only. Set the maximum number of results to retrieve on the Query .
consumer.delay	500	Option for HibernateConsumer only. Delay in milliseconds between each poll.
consumer.initialDelay	1000	Option for HibernateConsumer only. Millis before polling starts.
consumer.userFixedDelay	false	Option for HibernateConsumer only. Set to true to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.

3.19. Hl7

The hl7 component is used for working with the HL7 MLLP protocol and the HL7 model using the HAPI library. This component supports the following:

• HL7 MLLP codec for Mina

• Agnostic data format using either plain String objects or HAPI HL7 model objects.

• Type Converter from/to HAPI and String

• HL7 DataFormat using HAPI library

- Even more ease-of-use as it's integrated well with the camel-mina component.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-hl7</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.19.1. HL7 MLLP protocol

HL7 is often used with the HL7 MLLP protocol that is a text based TCP socket based protocol. This component ships with a Mina Codec that conforms to the MLLP protocol so you can easily expose a HL7 listener that accepts HL7 requests over the TCP transport. To expose a HL7 listener service we reuse the existing camel-mina component where we just use `HL7MLLPCodec` as codec.

The HL7 MLLP codec has the following options:

Name	Default Value	Description
startByte	0x0b	The start byte spanning the HL7 payload.
endByte1	0x1c	The first end byte spanning the HL7 payload.
endByte2	0x0d	The 2nd end byte spanning the HL7 payload.
charset	JVM Default	The encoding (is a charset name) to use for the codec.
convertLFtoCR	true	Will convert \n to \r (0x0d, 13 decimal) as HL7 usually uses \r as segment terminators. The HAPI library requires the use of \r.
validate	true	Whether HAPI Parser should validate or not.

3.19.1.1. Exposing a HL7 listener

In our Spring XML file, we configure an endpoint to listen for HL7 requests using TCP:

```
<endpoint id="hl7listener"
    uri="mina:tcp://localhost:8888?sync=true&codec=#hl7codec"/>
```

Notice we configure it to use camel-mina with TCP on the localhost on port 8888. We use `sync=true` to indicate that this listener is synchronous and therefore will return a HL7 response to the caller. Then we setup mina to use our HL7 codec with codec=#hl7codec. Notice that hl7codec is just a Spring bean ID, so we could have named it mygreatcodecforhl7 or whatever. The codec is also set up in the Spring XML file:

```
<bean id="hl7codec" class="org.apache.camel.component.hl7.HL7MLLPCodec">
    <property name="charset" value="iso-8859-1"/>
</bean>
```

And here we configure the charset encoding to use, and iso-8859-1 is commonly used.

The endpoint hl7listener can then be used in a route as a consumer, as this java DSL example illustrates:

```
from("hl7socket").to("patientLookupService");
```

This is a very simple route that will listen for HL7 and route it to a service named patientLookupService that is also a Spring bean ID we have configured in the Spring XML as:

```
<bean id="patientLookupService"
    class="com.mycompany.healtcare.service.PatientLookupService"/>
```

Another powerful feature of Camel is that we can have our business logic in POJO classes that is not tied to Camel as shown here:

```
public class PatientLookupService {
    public Message lookupPatient(Message input) throws HL7Exception {
        QRD qrd = (QRD)input.get("QRD");
        String patientId =
            qrd.getWhoSubjectFilter(0).getIDNumber().getValue();

        // find patient data based on the patient id and
        // create a HL7 model object with the response
        Message response = ... create and set response data
        return response
    }
```

Notice that this class uses just imports from the HAPI library and none from Camel.

3.19.2. HL7 Model using java.lang.String

The HL7MLLP codec uses plain Strings as its data format. Camel uses its Type Converter to convert to/from strings to the HAPI HL7 model objects. However, you can use plain String objects if you prefer, for instance if you wish to parse the data yourself.

3.19.3. HL7 Model using HAPI

The HL7 model uses Java objects from the HAPI library. Using this library, we can encode and decode from the EDI format (ER7) that is mostly used with HL7. With this model you can code with Java objects instead of the EDI based HL7 format that can be hard for humans to read and understand.

The ER7 sample below is a request to lookup a patient with the patient ID, 0101701234.

```
MSH|^~\\&|MYSENDER|MYRECEIVER|MYAPPLICATION||200612211200
||QRY^A19|1234|P|2.4
QRD|200612211200|R|I|GetPatient|||1^RD|0101701234|DEM||
```

Using the HL7 model we can work with the data as a ca.uhn.hl7v2.model.Message.Message object. To retrieve the patient ID for the patient in the ER7 above, you can do this in Java code:

```
Message msg = exchange.getIn().getBody(Message.class);
QRD qrd = (QRD)msg.get("QRD");
String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue();
```

Camel has built-in type converters, so when this operation is invoked:

```
Message msg = exchange.getIn().getBody(Message.class);
```

Camel will convert the received HL7 data from String to Message. This is powerful when combined with the HL7 listener, then you as the end-user don't have to work with byte[], String or any other simple object formats. You can just use the HAPI HL7 model objects.

3.19.4. Message Headers

The unmarshal operation adds these MSH fields as headers on the Camel message:

Key	MSH field	Example
`CamelHL7SendingApplication`	MSH-3	MYSERVER
`CamelHL7SendingFacility`	MSH-4	MYSERVERAPP
`CamelHL7ReceivingApplication`	MSH-5	MYCLIENT
`CamelHL7ReceivingFacility`	MSH-6	MYCLIENTAPP
`CamelHL7Timestamp`	MSH-7	20071231235900
`CamelHL7Security`	MSH-8	null
`CamelHL7MessageType`	MSH-9-1	ADT
`CamelHL7TriggerEvent`	MSH-9-2	A01
`CamelHL7MessageControl`	MSH-10	1234
`CamelHL7ProcessingId`	MSH-11	P
`CamelHL7VersionId`	MSH-12	2.4

3.19.5. Options

The HL7 Data Format supports the following options:

Option	Default	Description
`validate`	true	Whether the HAPI Parser should validate.

3.19.6. Dependencies

To use HL7 in your Camel routes you'll need to add a Maven dependency on camel-hl7 listed above, which implements this data format. The HAPI library is split into a base library and several structures libraries, one for each HL7v2 message version.

By default camel-hl7 only references the HAPI base library. Applications are responsible for including structures libraries themselves. For example, if a application works with HL7v2 message versions 2.4 and 2.5 then the following dependencies must be added:

```
<dependency>
    <groupId>ca.uhn.hapi</groupId>
    <artifactId>hapi-structures-v24</artifactId>
    <version>1.0</version>
</dependency>

<dependency>
    <groupId>ca.uhn.hapi</groupId>
    <artifactId>hapi-structures-v25</artifactId>
    <version>1.0</version>
</dependency>
```

Alternatively, an OSGi bundle containing the base library, all structure libraries and required dependencies (on the bundle classpath) can be downloaded from the HAPI Maven repository:

```
<dependency>
    <groupId>ca.uhn.hapi</groupId>
    <artifactId>hapi-osgi-base</artifactId>
    <version>1.0.1</version>
</dependency>
```

See the Camel Website for examples of this component in use.

3.20. HTTP4

The **http4:** component provides HTTP based endpoints for consuming external HTTP resources (as a client to call external servers using HTTP).

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-http4</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```


3.20.1. URI format

```
http4:hostname[:port][/resourceUri][?options]
```

This will by default use port 80 for HTTP and 443 for HTTPS.

You can append query options to the URI in the following format, `?option=value&option=value&...`

> ⚠️ **camel-http4 vs camel-jetty**
>
> You can produce only to endpoints generated by the HTTP4 component. Therefore it should never be used as input into your Camel routes. To bind/expose an HTTP endpoint via a HTTP server as input to a Camel route, use the Jetty Component instead.

3.20.2. HttpEndpoint Options

Name	Default Value	Description
x509HostnameVerifier	*See Description*	*Default value:* `org.apache.http.conn.ssl.BrowserCompatHostnameVerifier` **Camel 2.7 onwards:** You can refer to a different `org.apache.http.conn.ssl.X509HostnameVerifier` instance in the Registry such as `org.apache.http.conn.ssl.StrictHostnameVerifier` or `org.apache.http.conn.ssl.AllowAllHostnameVerifier`.

Name	Default Value	Description
throwExceptionOnFailure	true	Option to disable throwing the HttpOperationFailedException in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.
bridgeEndpoint	false	If true, HttpProducer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for requests. You may also set the **throwExceptionOnFailure** to be false to let the HttpProducer send all the fault response back. Also if set to true HttpProducer and CamelServlet will skip the gzip processing if the content-encoding is "gzip".
disableStreamCache	false	DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is false to support multiple reads, otherwise DefaultHttpBinding will set the request input stream directly in the message body.
httpBindingRef	null	Reference to a Camel HttpBinding object in the Registry. Recommended to use the httpBinding option instead.
httpBinding	null	To use a custom HttpBinding.
httpClientConfigurerRef	null	Reference to a Camel HttpClientConfigurer object in the Registry. Recommended to use the httpClientConfigurer option instead.
httpContext	null	To use a custom HttpContext when executing requests.
httpContextRef	null	Reference to a custom org.apache.http.protocol.HttpContext in the Registry. Recommended to use the httpContext option instead.
httpClientConfigurer	null	Reference to a org.apache.camel.component.http. HttpClientConfigurer in the Registry.
httpClient.XXX	null	Setting options on the BasicHttpParams. For instance httpClient.soTimeout=5000 will set the SO_TIMEOUT to 5 seconds. Look on the setter methods of the following parameter beans for a complete reference: AuthParamBean, ClientParamBean, ConnConnectionParamBean, ConnRouteParamBean, CookieSpecParamBean, HttpConnectionParamBean and HttpProtocolParamBean
clientConnectionManager	null	To use a custom org.apache.http.conn. ClientConnectionManager.
transferException	false	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a application/x-java-serialized-object content type (for example using Section 3.24, "Jetty" or Section 3.40, "Servlet" Camel components). On the producer side the exception will be deserialized and thrown as is, instead of the HttpOperationFailedException. The caused exception is required to be serialized.
maxTotalConnections	200	The maximum number of connections.
connectionsPerRoute	20	The maximum number of connections per route.

The following authentication options can also be set on the HttpEndpoint:

3.20.2.1. Setting Basic Authentication and Proxy

Name	Default Value	Description
username	null	Username for authentication.
password	null	Password for authentication.
domain	null	The domain name for authentication.
host	null	The host name authentication.
proxyHost	null	The proxy host name
proxyPort	null	The proxy port number
proxyUsername	null	Username for proxy authentication
proxyPassword	null	Password for proxy authentication
proxyDomain	null	The proxy domain name
proxyNtHost	null	The proxy Nt host name

3.20.3. HttpComponent Options

Name	Default Value	Description
httpBinding	null	To use a custom `org.apache.camel.component.http.HttpBinding`.
httpClientConfigurer	null	To use a custom `org.apache.camel.component.http.HttpClientConfigurer`.
httpConnectionManager	null	To use a custom `org.apache.commons.httpclient.HttpConnectionManager`.
httpContext	null	To use a custom HttpContext when executing requests.
x509HostnameVerifier	null	**Camel 2.7 onwards:** To use a custom `org.apache.http.conn.ssl.X509HostnameVerifier`

3.20.4. Message Headers

Name	Type	Description
Exchange.HTTP_URI	String	URI to call. This will override existing URI set directly on the endpoint.
Exchange.HTTP_PATH	String	Request URI's path, the header will be used to build the request URI with the HTTP_URI.
Exchange.HTTP_QUERY	String	URI parameters. This will override existing URI parameters set directly on the endpoint.
Exchange.HTTP_RESPONSE_CODE	int	The HTTP response code from the external server. Is 200 for OK.
Exchange.HTTP_CHARACTER_ENCODING	String	Character encoding.

Name	Type	Description
Exchange.CONTENT_TYPE	String	The HTTP content type. Is set on both the IN and OUT message to provide a content type, such as `text/html`.
Exchange.CONTENT_ENCODING	String	The HTTP content encoding. Is set on both the IN and OUT message to provide a content encoding, such as `gzip`.

3.20.5. Message Body

Camel will store the HTTP response from the external server on the OUT body. All headers from the IN message will be copied to the OUT message, so headers are preserved during routing. Additionally Camel will add the HTTP response headers as well to the OUT message headers.

3.20.6. Response code

Camel will handle according to the HTTP response code:

- Response code is in the range 100..299, Camel regards it as a success response.

- Response code is in the range 300..399, Camel regards it as a redirection response and will throw a `HttpOperationFailedException` with the information.

- Response code is 400+, Camel regards it as an external server failure and will throw a `HttpOperationFailedException` with the information.

> **throwExceptionOnFailure**
>
> The option, `throwExceptionOnFailure`, can be set to `false` to prevent the `HttpOperationFailedException` from being thrown for failed response codes. This allows you to get any response from the remote server. There is a sample below demonstrating this.

3.20.7. HttpOperationFailedException

This exception contains the following information:

- The HTTP status code

- The HTTP status line (text of the status code)

- Redirect location, if server returned a redirect

- Response body as a `java.lang.String`, if server provided a body as response

3.20.8. Calling using GET or POST

The following algorithm is used to determine whether the `GET` or `POST` HTTP method should be used: 1. Use method provided in header. 2. `GET` if query string is provided in header. 3. `GET` if endpoint is configured with a query string. 4. `POST` if there is data to send (body is not `null`). 5. `GET` otherwise.

3.20.9. How to get access to HttpServletRequest and HttpServletResponse

You can get access to these two using the Camel type converter system using **NOTE** You can get the request and response not just from the processor after the camel-jetty or camel-cxf endpoint.

```
HttpServletRequest request = exchange.getIn().getBody(
    HttpServletRequest.class);
HttpServletRequest response =
      exchange.getIn().getBody(HttpServletResponse.class);
```

3.20.10. Configuring URI to call

You can set the HTTP producer's URI directly form the endpoint URI. In the route below, Camel will call out to the external server, `oldhost`, using HTTP.

```
from("direct:start").to("http4://oldhost");
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <to uri="http4://oldhost"/>
  </route>
</camelContext>
```

You can override the HTTP endpoint URI by adding a header with the key, `HttpConstants.HTTP_URI`, on the message.

```
from("direct:start")
   .setHeader(HttpConstants.HTTP_URI, constant("http://newhost"))
   .to("http4://oldhost");
```

In the sample above Camel will call the http://newhost despite the fact the endpoint is configured with http4:// oldhost. where Constants is the class, `org.apache.camel.component.http4.Constants`.

3.20.11. Configuring URI Parameters

The **http** producer supports URI parameters to be sent to the HTTP server. The URI parameters can either be set directly on the endpoint URI or as a header with the key `Exchange.HTTP_QUERY` on the message.

```
from("direct:start").to("http4://oldhost?order=123&detail=short");
```

Or options provided in a header:

```
from("direct:start")
   .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
   .to("http4://oldhost");
```

3.20.12. How to set the http method (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE) to the HTTP producer

The HTTP4 component provides a way to set the HTTP request method by setting the message header. Here is an example;

```
from("direct:start")
    .setHeader(Exchange.HTTP_METHOD,
          constant(org.apache.camel.component.http4.HttpMethods.POST))
    .to("http4://www.google.com")
    .to("mock:results");
```

The method can be written a bit shorter using the string constants:

```
.setHeader("CamelHttpMethod", constant("POST"))
```

And the equivalent Spring sample:

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
    <route>
        <from uri="direct:start"/>
        <setHeader headerName="CamelHttpMethod">
           <constant>POST</constant>
        </setHeader>
        <to uri="http4://www.google.com"/>
        <to uri="mock:results"/>
    </route>
</camelContext>
```

3.20.13. Configuring a Proxy

The HTTP4 component provides a way to configure a proxy.

```
from("direct:start")
      .to("http4://oldhost?proxyHost=www.myproxy.com&proxyPort=80");
```

There is also support for proxy authentication via the `proxyUsername` and `proxyPassword` options.

3.20.13.1. Using proxy settings outside of URI

To avoid System properties conflicts, you can set proxy configuration only from the CamelContext or URI. Java DSL:

```
context.getProperties().put("http.proxyHost", "172.168.18.9");
 context.getProperties().put("http.proxyPort" "8080");
```

Spring XML

```
<camelContext>
      <properties>
           <property key="http.proxyHost" value="172.168.18.9"/>
```

```
        <property key="http.proxyPort" value="8080"/>
    </properties>
  </camelContext>
```

Camel will first set the settings from Java System or CamelContext Properties and then the endpoint proxy options if provided. So you can override the system properties with the endpoint options.

3.20.14. Configuring charset

If you are using `POST` to send data you can configure the `charset` using the `Exchange` property:

```
exchange.setProperty(Exchange.CHARSET_NAME, "ISO-8859-1");
```

3.20.14.1. Sample with scheduled poll

This sample polls the Google homepage every 10 seconds and write the page to the file `message.html`:

```
from("timer://foo?fixedRate=true&delay=0&period=10000")
    .to("http4://www.google.com")
    .setHeader(FileComponent.HEADER_FILE_NAME, "message.html")
    .to("file:target/google");
```

3.20.14.2. URI Parameters from the endpoint URI

In this sample we have the complete URI endpoint that is just what you would have typed in a web browser. Multiple URI parameters can of course be set using the `&` character as separator, just as you would in the web browser. Camel does no tricks here.

```
// we query for Camel at the Google page
template.sendBody("http4://www.google.com/search?q=Camel", null);
```

3.20.14.3. URI Parameters from the Message

```
Map headers = new HashMap();
headers.put(HttpProducer.QUERY, "q=Camel&lr=lang_en");
// we query for Camel and English language at Google
template.sendBody("http4://www.google.com/search", null, headers);
```

In the header value above notice that it should **not** be prefixed with ? and you can separate parameters as usual with the `&` char.

3.20.14.4. Getting the Response Code

You can get the HTTP response code from the HTTP4 component by getting the value from the Out message header with `HttpProducer.HTTP_RESPONSE_CODE`.

```
Exchange exchange =
      template.send("http4://www.google.com/search", new Processor() {
```

```
    public void process(Exchange exchange) throws Exception {
        exchange.getIn().setHeader(
            HttpProducer.QUERY, constant("hl=en&q=activemq"));
    }
});
Message out = exchange.getOut();
int responseCode = out.getHeader(HttpProducer.HTTP_RESPONSE_CODE,
    Integer.class);
```

3.20.15. Disabling Cookies

To disable cookies you can set the HTTP Client to ignore cookies by adding this URI option: `httpClient.cookiePolicy=ignoreCookies`

3.20.16. Advanced Usage

If you need more control over the HTTP producer you should use the `HttpComponent` where you can set various classes to give you custom behavior.

3.20.16.1. Setting up SSL for HTTP Client

Basically camel-http4 component is built on the top of Apache HTTP client. Please refer to SSL/TLS customization for details or have a look into the `org.apache.camel.component.http4.HttpsServerTestSupport` unit test base class. You can also implement a custom `org.apache.camel.component.http4.HttpClientConfigurer` to do some configuration on the http client if you need full control of it.

However if you *just* want to specify the keystore and truststore you can do this with Apache HTTP `HttpClientConfigurer`, for example:

```
KeyStore keystore = ...;
KeyStore truststore = ...;

SchemeRegistry registry = new SchemeRegistry();
registry.register(new Scheme("https", 443, new SSLSocketFactory(
    keystore, "mypassword", truststore)));
```

And then you need to create a class that implements `HttpClientConfigurer`, and registers https protocol providing a keystore or truststore per example above. Then, from your Camel route builder class you can hook it up like so:

```
HttpComponent httpComponent =
    getContext().getComponent("http4", HttpComponent.class);
httpComponent.setHttpClientConfigurer(new MyHttpClientConfigurer());
```

If you are doing this using the Spring DSL, you can specify your `HttpClientConfigurer` using the URI. For example:

```
<bean id="myHttpClientConfigurer"
 class="my.https.HttpClientConfigurer">
</bean>
```

```
<to uri="https4://myhostname.com:443/myURL?httpClientConfigurer=
myHttpClientConfigurer"/>
```

As long as you implement the HttpClientConfigurer and configure your keystore and truststore as described above, it will work fine.

3.21. Jasypt

Jasypt is a simplified encryption library which makes encryption and decryption easy. Camel integrates with Jasypt to allow sensitive information in Section 3.34, "Properties" files to be encrypted. By dropping **camel-jasypt** on the classpath those encrypted values will automatic be decrypted on-the-fly by Camel. This ensures that human eyes can't easily spot sensitive information such as usernames and passwords.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jasypt</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```


3.21.1. Tooling

The Section 3.21, "Jasypt" component provides a little command line tooling to encrypt or decrypt values.

The console output the syntax and which options it provides:

```
Apache Camel Jasypt takes the following options

  -h or -help = Displays the help screen
  -c or -command <command> = Command either encrypt or decrypt
  -p or -password <password> = Password to use
  -i or -input <input> = Text to encrypt or decrypt
  -a or -algorithm <algorithm> = Optional algorithm to use
```

For example to encrypt the value tiger you run with the following parameters. In the apache Camel kit, you cd into the lib folder and run the following java cmd, where <*CAMEL_HOME*> is where you have downloaded and extract the Camel distribution.

```
$ cd <CAMEL_HOME>/lib
$ java -jar camel-jasypt-2.5.0.jar -c encrypt -p secret -i tiger
```

Which outputs the following result

```
Encrypted text: qaEEacuW7BUti8LcMgyjKw==
```

This means the encrypted representation qaEEacuW7BUti8LcMgyjKw== can be decrypted back to tiger if you know the master password which was secret . If you run the tool again then the encrypted value will return a different result. But decrypting the value will always return the correct original value.

So you can test it by running the tooling using the following parameters:

```
$ cd <CAMEL_HOME>/lib
$ java -jar camel-jasypt-2.5.0.jar -c decrypt -p secret
      -i qaEEacuW7BUti8LcMgyjKw==
```

Which outputs the following result:

```
Decrypted text: tiger
```

The idea is then to use those encrypted values in your Section 3.34, "Properties" files. Notice how the password value is encrypted and the value has the tokens surrounding ENC(value here)

```
# refer to a mock endpoint name by that encrypted password
cool.result=mock:{{cool.password}}

# here is a password which is encrypted
cool.password=ENC(bsW9uV37gQ0QHFu7KO03Ww==)
```

3.21.1.1. Tooling dependencies for Camel 2.6

The tooling requires the following JARs in the classpath, which has been enlisted in the MANIFEST.MF file of camel-jasypt with optional/ as prefix. Hence why the java cmd above can pickup the needed JARs from the Apache Distribution in the optional directory.

```
jasypt-1.6.jar commons-lang-2.4.jar commons-codec-1.4.jar icu4j-4.0.1.jar
```

> ⚠️ **Java 1.5 users**
>
> The icu4j-4.0.1.jar is only needed when running on JDK 1.5.
>
> This JAR is not distributed by Apache Camel and you have to download it manually and copy it to the lib/optional directory of the Camel distribution. You can download it from Apache Central Maven repo .

3.21.1.2. Tooling dependencies for Camel 2.7 onwards

Jasypt 1.7 onwards is now fully standalone so no additional JARs is needed.

3.21.2. URI Options

The options below are exclusive for the Section 3.21, "Jasypt" component.

Name	Default Value	Type	Description
password	null	String	Specifies the master password to use for decrypting. This option is mandatory. See below for more details.
algorithm	null	String	Name of an optional algorithm to use.

3.21.3. Protecting the master password

The master password used by Section 3.21, "Jasypt" must be provided, so it is capable of decrypting the values. However having this master password out in the open may not be an ideal solution. Therefore you could for

example provide it as a JVM system property or as a OS environment setting. If you decide to do so then the `password` option supports prefixes which dictates this. `sysenv:` means to lookup the OS system environment with the given key. `sys:` means to lookup a JVM system property.

For example you could provided the password before you start the application

```
$ export CAMEL_ENCRYPTION_PASSWORD=secret
```

Then start the application, such as running the start script.

When the application is up and running you can unset the environment

```
$ unset CAMEL_ENCRYPTION_PASSWORD
```

The password option is then a matter of defining as follows: `password=sysenv:CAMEL_ENCRYPTION_PASSWORD`.

3.21.4. Example with Java DSL

In Java DSL you need to configure Section 3.21, "Jasypt" as a `JasyptPropertiesParser` instance and set it on the Section 3.34, "Properties" component as shown below:

```
// create the jasypt properties parser
JasyptPropertiesParser jasypt = new JasyptPropertiesParser();
// and set the master password
jasypt.setPassword("secret");

// create the properties component
PropertiesComponent pc = new PropertiesComponent();
pc.setLocation(
    "classpath:org/apache/camel/component/jasypt/myproperties.properties");
// and use the jasypt properties parser so we can decrypt values
pc.setPropertiesParser(jasypt);

// add properties component to Camel context
context.addComponent("properties", pc);
```

The properties file `myproperties.properties` then contain the encrypted value, such as shown below. Notice how the password value is encrypted and the value has the tokens surrounding `ENC(value here)`

```
# refer to a mock endpoint name by that encrypted password
cool.result=mock:{{cool.password}}

# here is a password which is encrypted
cool.password=ENC(bsW9uV37gQ0QHFu7KO03Ww==)
```

3.21.5. Example with Spring XML

In Spring XML you need to configure the `JasyptPropertiesParser` which is shown below. Then the Camel Section 3.34, "Properties" component is told to use `jasypt` as the properties parser, which means Section 3.21, "Jasypt" have its chance to decrypt values looked up in the properties.

```
<!-- define the jasypt properties parser with the given password -->
```

```
<bean id="jasypt"
      class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <property name="password" value="secret"/>
</bean>

<!-- define the Camel properties component -->
<bean id="properties"
      class="org.apache.camel.component.properties.PropertiesComponent">
    <!-- the properties file is in the classpath -->
    <property name="location"
          value=
        "classpath:org/apache/camel/component/jasypt/myprops.properties"/>
    <!-- and let it leverage the jasypt parser -->
    <property name="propertiesParser" ref="jasypt"/>
</bean>
```

The Section 3.34, "Properties" component can also be inlined inside the `<camelContext>` tag which is shown below. Notice how we use the `propertiesParserRef` attribute to refer to Section 3.21, "Jasypt" .

```
<!-- define the jasypt properties parser with the given password -->
<bean id="jasypt"
      class="org.apache.camel.component.jasypt.JasyptPropertiesParser">
    <!-- password is mandatory, you can prefix it with sysenv: or sys:
      to indicate it should use an OS environment or JVM system property
      value, so you don't have the master password defined here -->
    <property name="password" value="secret"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- define the Camel properties placeholder and let it use jasypt -->
    <propertyPlaceholder
       id="properties"
       location=
          "classpath:org/apache/camel/component/jasypt/myproperties.properties"
          propertiesParserRef="jasypt"/>
    <route>
       <from uri="direct:start"/>
       <to uri="{{cool.result}}"/>
    </route>
</camelContext>
```

3.22. JCR

The **jcr** component allows you to add nodes to a JCR (JSR-170) compliant content repository (for example, Apache Jackrabbit).

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jcr</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.22.1. URI format

```
jcr://user:password@repository/path/to/node
```

3.22.2. Usage

The `repository` element of the URI is used to look up the JCR `Repository` object in the Camel context registry.

If a message is sent to a JCR producer endpoint:

• A new node is created in the content repository,

• All the message properties of the IN message are transformed to JCR `Value` instances and added to the new node,

• The node's UUID is returned in the OUT message.

3.22.3. Message properties

All message properties are converted to node properties, except for the `CamelJcrNodeName` property (you can refer to `JcrConstants.NODE_NAME` in your code), which is used to determine the node name.

3.22.4. Example

The snippet below creates a node named `node` under the `/home/test` node in the content repository. One additional attribute is added to the node as well: `my.contents.property` which will contain the body of the message being sent.

```
from("direct:a").setProperty(JcrConstants.JCR_NODE_NAME, constant("node"))
    .setProperty("my.contents.property", body())
    .to("jcr://user:pass@repository/home/test");
```

3.23. JDBC

The **jdbc** component enables you to access databases through JDBC, where SQL queries and operations are sent in the message body. This component uses the standard JDBC API, unlike the Section 3.46, "SQL Component" component, which uses spring-jdbc.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jdbc</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

> **Warning**
>
> This component can only be used to define producer endpoints, which means that you cannot use the JDBC component in a `from()` statement.

> **Note**
>
> This component can not be used as a Transactional Client. If you need transaction support in your route, you should use the Section 3.46, "SQL Component" component instead.

3.23.1. URI format

```
jdbc:dataSourceName[?options]
```

This component only supports producer endpoints.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.23.2. Options

Name	Default Value	Description
`readSize`	`0`	The default maximum number of rows that can be read by a polling query. The default value is 0.
`statement.<xxx>`	`null`	Sets additional options on the `java.sql.Statement` that is used behind the scenes to execute the queries. For instance, `statement.maxRows=10` . For detailed documentation, see the `java.sql.Statement` javadoc documentation.
`useJDBC4ColumnNameAnd-LabelSemantics`	`true`	Sets whether to use JDBC 4/3 column label/name semantics. You can use this option to turn it `false` in case you have issues with your JDBC driver to select data. This only applies when using `SQL SELECT` using aliases (for example, `SQL SELECT id as identifier, name as given_name from persons`).
`resetAutoCommit`	`true`	Camel will set the autoCommit on the JDBC connection to be false, commit the change after executing the statement and reset the autoCommit flag of the connection at the end, if the resetAutoCommit is true. If the JDBC connection doesn't support resetting the autoCommit flag, you can set the resetAutoCommit flag to be false, and Camel will not try to reset the autoCommit flag.

3.23.3. Result

The result is returned in the OUT body as an `ArrayList<HashMap<String, Object>>`. The `List` object contains the list of rows and the `Map` objects contain each row with the `String` key as the column name.

Note: This component fetches `ResultSetMetaData` to be able to return the column name as the key in the `Map`.

3.23.3.1. Message Headers

Header	Description
`CamelJdbcRowCount`	If the query is a `SELECT`, query the row count is returned in this OUT header.
`CamelJdbcUpdateCount`	If the query is an `UPDATE`, query the update count is returned in this OUT header.
`CamelGeneratedKeysRows`	Rows that contain the generated keys. If you insert data using SQL INSERT, setting this value to true causes the generated keys to be returned in headers.
`CamelGeneratedKeys-RowCount`	The number of rows in the header that contains generated keys.

3.23.4. Samples

In the following example, we fetch the rows from the customer table.

First we register our datasource in the Camel registry as `testdb`:

```
JndiRegistry reg = super.createRegistry();
reg.bind("testdb", ds);
return reg;
```

Then we configure a route that routes to the JDBC component, so the SQL will be executed. Note how we refer to the `testdb` datasource that was bound in the previous step:

```
// let's add a simple route
public void configure() throws Exception {
    from("direct:hello").to("jdbc:testdb?readSize=100");
}
```

Or you can create a `DataSource` in Spring like this:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
     <from uri="timer://kickoff?period=10000"/>
     <setBody>
       <constant>select * from customer</constant>
     </setBody>
     <to uri="jdbc:testdb"/>
     <to uri="mock:result"/>
  </route>
</camelContext>

<!-- Just add a demo to show how to
    bind a date source for Camel in Spring-->
<bean id="testdb"
```

```
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:mem:camel_jdbc" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>
```

We create an endpoint, add the SQL query to the body of the IN message, and then send the exchange. The result of the query is returned in the OUT body:

```
// first we create our exchange using the endpoint
Endpoint endpoint = context.getEndpoint("direct:hello");
Exchange exchange = endpoint.createExchange();
// then we set the SQL on the in body
exchange.getIn().setBody("select * from customer order by ID");

// now we send the exchange to the endpoint, and receive Camel response
Exchange out = template.send(endpoint, exchange);

// assertions of the response
assertNotNull(out);
assertNotNull(out.getOut());
ArrayList<HashMap<String, Object>> data = out.getOut().getBody(
    ArrayList.class);
assertNotNull("out body could not be converted to an ArrayList - was: "
    + out.getOut().getBody(), data);
assertEquals(2, data.size());
HashMap<String, Object> row = data.get(0);
assertEquals("cust1", row.get("ID"));
assertEquals("jbloggs", row.get("NAME"));
row = data.get(1);
assertEquals("cust2", row.get("ID"));
assertEquals("nsandhu", row.get("NAME"));
```

If you want to work on the rows one by one instead of the entire ResultSet at once you need to use the Section 2.48, "Splitter" EIP such as:

```
from("direct:hello")
        // here we split the data from the testdb into new messages one by one
        // so the mock endpoint will receive a message per row in the table
    .to("jdbc:testdb").split(body()).to("mock:result");
```

3.24. Jetty

The **jetty** component provides HTTP-based endpoints for consuming HTTP requests. That is, the Jetty component behaves as a simple Web server. Jetty can also be used as a http client which mean you can also use it with Camel as a Producer.

Note Jetty is stream based, which means the input it receives is submitted to Camel as a stream. That means you will only be able to read the content of the stream once. If you find a situation where the message body appears to be empty or you need to access the data multiple times (for example,: doing multicasting, or redelivery error handling) you should use Stream Caching or convert the message body to a String which is safe to be re-read multiple times.

3.24.1. URI format

```
jetty:http://hostname[:port][/resourceUri][?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.24.2. Options

Name	Default Value	Description
`sessionSupport`	`false`	Specifies whether to enable the session manager on the server side of Jetty.
`httpClient.XXX`	`null`	Configuration of Jetty's HttpClient . For example, setting `httpClient.idleTimeout=30000` sets the idle timeout to 30 seconds.
`httpBindingRef`	`null`	Reference to an Camel `HttpBinding` object in the Registry . `HttpBinding` can be used to customize how a response should be written.
`jettyHttpBindingRef`	`null`	Reference to a Camel `JettyHttpBinding` object in the Registry . `JettyHttpBinding` can be used to customize how a response should be written.
`matchOnUriPrefix`	`false`	Whether or not the `CamelServlet` should try to find a target consumer by matching the URI prefix if no exact match is found. See here How do I let Jetty match wildcards .
`handlers`	`null`	Specifies a comma-delimited set of `org.mortbay.jetty.Handler` instances in your Registry (such as your Spring `ApplicationContext`). These handlers are added to the Jetty servlet context (for example, to add security).
`chunked`	`true`	If this option is false Jetty servlet will disable the HTTP streaming and set the content-length header on the response
`enableJmx`	`false`	If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.
`disableStreamCache`	`false`	Determines whether or not the raw input stream from Jetty is cached or not (Camel will read the stream into a in memory/ overflow to file, Stream Caching) cache. By default Camel will cache the Jetty input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to `true` when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. DefaultHttpBinding will copy the request input stream into a stream cache and put it into message body if this option is `false` to support reading the stream multiple times. If you use [Jetty] to bridge/proxy an endpoint then consider enabling this option to improve performance, in case you do not need to read the message payload multiple times.
`bridgeEndpoint`	`false`	If the option is true, HttpProducer will ignore the Exchange.HTTP_URI header, and use the endpoint's URI for request. You may also set the **throwExceptionOnFailure** to be false to let the HttpProducer send all the fault response

Name	Default Value	Description
		back. If the option is true, HttpProducer and CamelServlet will skip the gzip processing if the content-encoding is "gzip". Also consider setting *disableStreamCache* to true to optimize when bridging.
enableMultipartFilter	true	Whether Jetty org.eclipse. jetty.servlets. MultiPartFilter is enabled or not. You should set this value to false when bridging endpoints, to ensure multipart requests is proxied/bridged as well.
multipartFilterRef	null	**Camel 2.6 onwards:** Allows using a custom multipart filter. Note: setting multipartFilterRef forces the value of enableMultipartFilter to true .
FiltersRef	null	Camel 2.9: Allows using a custom filter which is put into a list and can be found in the Registry
sslContextParametersRef	null	Camel 2.8: Reference to an org.apache.camel.util.jsse.SSLContextParameters object in the Camel Registry. This reference overrides any configured SSLContextParameters at the component level.
traceEnabled	false	Specifies whether to enable HTTP TRACE for this Jetty consumer. By default TRACE is turned off.
continuationTimeout	null	**Camel 2.6 onwards:** Allows to set a timeout in milliseconds when using Section 3.24, "Jetty" as consumer (server). By default Jetty uses 30000. You can use a value of <= 0 to never expire. If a timeout occurs then the request will be expired and Jetty will return back a http error 503 to the client. This option is only in use when using Section 3.24, "Jetty" with the Asynchronous Routing Engine .
useContinuation	true	**Camel 2.6 onwards:** Whether or not to use Jetty continuations for the Jetty Server.

3.24.3. Message Headers

Camel uses the same message headers as the Section 3.20, "HTTP4" component. It also uses (Exchange.HTTP_CHUNKED,CamelHttpChunked) header to turn on or turn off the chunked encoding on the camel-jetty consumer.

Camel also populates **all** request.parameter and request.headers. For example, given a client request with the URL, http://myserver/myserver?orderid=123 , the exchange will contain a header named orderid with the value 123.

You can get the request.parameter from the message header not only from Get Method, but also other HTTP methods.

3.24.4. Usage

The Jetty component only supports consumer endpoints. Therefore a Jetty endpoint URI should be used only as the **input** for a Camel route (in a from() DSL call). To issue HTTP requests against other HTTP endpoints, use the HTTP4 Component

3.24.5. Component Options

The `JettyHttpComponent` provides the following options:

Name	Default Value	Description
enableJmx	false	If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.
sslKeyPassword	null	**Consumer only** : The password for the keystore when using SSL.
sslPassword	null	**Consumer only** : The password when using SSL.
sslKeystore	null	**Consumer only** : The path to the keystore.
minThreads	null	**Consumer only** : To set a value for minimum number of threads in server thread pool.
maxThreads	null	**Consumer only** : To set a value for maximum number of threads in server thread pool.
threadPool	null	**Consumer only** : To use a custom thread pool for the server.
sslSocketConnectors	null	**Consumer only:** A map which contains per port number specific SSL connectors. See section *SSL support* for more details.
socketConnectors	null	**Consumer only:** A map which contains per port number specific HTTP connectors. Uses the same principle as `sslSocketConnectors` and therefore see section *SSL support* for more details.
sslSocketConnector-Properties	null	**Consumer only**. A map which contains general SSL connector properties. See section *SSL support* for more details.
socketConnectorProperties	null	**Consumer only**. A map which contains general HTTP connector properties. Uses the same principle as `sslSocketConnectorProperties` and therefore see section *SSL support* for more details.
httpClient	null	**Producer only** : To use a custom `HttpClient` with the jetty producer.
httpClientMinThreads	null	**Producer only** : To set a value for minimum number of threads in `HttpClient` thread pool.
httpClientMaxThreads	null	**Producer only** : To set a value for maximum number of threads in `HttpClient` thread pool.
httpClientThreadPool	null	**Producer only** : To use a custom thread pool for the client.
sslContextParameters	null	To configure a custom SSL/TLS configuration options at the component level.

3.24.6. Sample

In this sample we define a route that exposes a HTTP service at `http://localhost:8080/myapp/myservice` :

```
from("jetty:http://localhost:{{port}}/myapp/myservice").process(
    new MyBookService());
```

Usage of localhost

When you specify `localhost` in a URL, Camel exposes the endpoint only on the local TCP/IP network interface, so it cannot be accessed from outside the machine it operates on.

If you need to expose a Jetty endpoint on a specific network interface, the numerical IP address of this interface should be used as the host. If you need to expose a Jetty endpoint on all network interfaces, the `0.0.0.0` address should be used.

Our business logic is implemented in the `MyBookService` class, which accesses the HTTP request contents and then returns a response. **Note:** The `assert` call appears in this example, because the code is part of an unit test.

```
public class MyBookService implements Processor {
    public void process(Exchange exchange) throws Exception {
        // just get the body as a string
        String body = exchange.getIn().getBody(String.class);
        // we have access to the HttpServletRequest here and we can grab it
        // if we need it
        HttpServletRequest req =
                exchange.getIn().getBody(HttpServletRequest.class);
        assertNotNull(req);

        // for unit testing
        assertEquals("bookid=123", body);

        // send a html response
        exchange.getOut().setBody(
            "<html><body>Book 123 is Factory Patterns</body></html>");
    }
}
```

The following sample shows a content-based route that routes all requests containing the URI parameter, `one`, to the endpoint, `mock:one`, and all others to `mock:other`.

```
from("jetty:" + serverUri)
    .choice()
    .when().simple("in.header.one").to("mock:one")
    .otherwise()
    .to("mock:other");
```

So if a client sends the HTTP request, `http://serverUri?one=hello` , the Jetty component will copy the HTTP request parameter, `one` to the exchange's `in.header`. We can then use the `Simple` language to route exchanges that contain this header to a specific endpoint and all others to another. If we used a language more powerful than Simple -- such as EL or OGNL --we could also test for the parameter value and do routing based on the header value as well.

3.24.7. Session Support

The session support option, `sessionSupport`, can be used to enable a `HttpSession` object and access the session object while processing the exchange. For example, the following route enables sessions:

```
<route>
    <from uri="jetty:http://0.0.0.0/myapp/myservice/?sessionSupport=true"/>
    <processRef ref="myCode"/>
<route>
```

The `myCode` Processor can be instantiated by a Spring `bean` element:

```
<bean id="myCode"class="com.mycompany.MyCodeProcessor"/>
```

where the processor implementation can access the `HttpSession` as follows:

```
public void process(Exchange exchange) throws Exception {
    HttpSession session = exchange.getIn(HttpMessage.class).getRequest()
        .getSession();
    ...
}
```

3.24.8. SSL Support (HTTPS)

The Jetty component supports SSL/TLS configuration through the Camel JSSE Configuration Utility This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the Jetty component.

Programmatic configuration of the component:

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

JettyComponent jettyComponent = getContext().getComponent("jetty",
    JettyComponent.class);
jettyComponent.setSslContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
...
<camel:sslContextParameters
id="sslContextParameters">
<camel:keyManagers
keyPassword="keyPassword">
<camel:keyStore
resource="/users/home/server/keystore.jks"
password="keystorePassword"/>
</camel:keyManagers>
</camel:sslContextParameters>...
...
<to uri="jetty:https://127.0.0.1/mail/?sslContextParametersRef=... \
    sslContextParameters"/>
...
```

You can also configure Jetty for SSL directly. In this case, simply format the URI with the `https://` prefix---for example:

```
<from uri="jetty:https://0.0.0.0/myapp/myservice/"/>
```

Jetty also needs to know where to load your keystore from and what passwords to use in order to load the correct SSL certificate. Set the following JVM System Properties:

- `org.eclipse.jetty.ssl.keystore` specifies the location of the Java keystore file, which contains the Jetty server's own X.509 certificate in a *key entry* . A key entry stores the X.509 certificate (effectively, the *public key*) and also its associated private key.

- `org.eclipse.jetty.ssl.password` the store password, which is required to access the keystore file (this is the same password that is supplied to the `keystore` command's `-storepass` option).

- `org.eclipse.jetty.ssl.keypassword` the key password, which is used to access the certificate's key entry in the keystore (this is the same password that is supplied to the `keystore` command's `-keypass` option).

For details of how to configure SSL on a Jetty endpoint, read the Jetty documentation here.

The value you use as keys in the above map is the port you configure Jetty to listen on.

3.24.8.1. Configuring general SSL properties

Instead of a per port number specific SSL socket connector (as shown above) you can now configure general properties which applies for all SSL socket connectors (which is not explicit configured as above with the port number as entry).

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
    <property name="sslSocketConnectorProperties">
        <properties>
            <property name="password"value="..."/>
            <property name="keyPassword"value="..."/>
            <property name="keystore"value="..."/>
            <property name="needClientAuth"value="..."/>
            <property name="truststore"value="..."/>
        </properties>
    </property>
</bean>
```

3.24.8.2. Configuring general HTTP properties

Instead of a per port number specific HTTP socket connector (as shown above) you can now configure general properties which applies for all HTTP socket connectors (which is not explicit configured as above with the port number as entry).

```
<bean id="jetty" class="org.apache.camel.component.jetty.JettyHttpComponent">
    <property name="socketConnectorProperties">
        <properties>
            <property name="acceptors" value="4"/>
            <property name="maxIdleTime" value="300000"/>
        </properties>
    </property>
</bean>
```

3.24.8.3. Default behavior for returning HTTP status codes

The default behavior of HTTP status codes is defined by the `org.apache.camel.component.http.DefaultHttpBinding` class, which handles how a response is written and also sets the HTTP status code.

If the exchange was processed successfully, the 200 HTTP status code is returned. If the exchange failed with an exception, the 500 HTTP status code is returned, and the stacktrace is returned in the body. If you want to specify which HTTP status code to return, set the code in the `HttpProducer.HTTP_RESPONSE_CODE` header of the OUT message.

3.24.8.4. Jetty JMX support

Camel-jetty supports the enabling of Jetty's JMX capabilities at the component and endpoint level with the endpoint configuration taking priority. Note that JMX must be enabled within the Camel context in order to enable JMX support in this component as the component provides Jetty with a reference to the MBeanServer registered with the Camel context. Because the camel-jetty component caches and reuses Jetty resources for a given protocol/host/port pairing, this configuration option will only be evaluated during the creation of the first endpoint to use a protocol/host/port pairing.

For example, given two routes created from the following XML fragments, JMX support would remain enabled for all endpoints listening on "https://0.0.0.0".

```
<from uri="jetty:https://0.0.0.0/myapp/myservice1/?enableJmx=true"/>
```

```
<from uri="jetty:https://0.0.0.0/myapp/myservice2/?enableJmx=false"/>
```

The camel-jetty component also provides for direct configuration of the Jetty MBeanContainer. Jetty creates MBean names dynamically. If you are running another instance of Jetty outside of the Camel context and sharing the same MBeanServer between the instances, you can provide both instances with a reference to the same MBeanContainer in order to avoid name collisions when registering Jetty MBeans.

3.25. JMS

Using ActiveMQ

If you are using Apache ActiveMQ, you should prefer the Section 3.1, "ActiveMQ" component as it has been optimized for it. . All of the options and samples on this page are also valid for the ActiveMQ component.

The JMS component allows messages to be sent to (or consumed from) a JMS Queue or Topic. The implementation of the JMS Component uses Spring's JMS support for declarative transactions, using Spring's `JmsTemplate` for sending and a `MessageListenerContainer` for consuming.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jms</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.25.1. URI format

```
jms:[queue:|topic:]destinationName[?options]
```

where `destinationName` is a JMS queue or topic name. By default, the `destinationName` is interpreted as a queue name. For example, to connect to the queue, FOO.BAR, use:

```
jms:FOO.BAR
```

You can include the optional `queue:` prefix, if you prefer:

```
jms:queue:FOO.BAR
```

To connect to a topic, you *must* include the `topic:` prefix. For example, to connect to the topic, `Stocks.Prices`, use:

```
jms:topic:Stocks.Prices
```

Append query options to the URI using the following format, `?option=value&option=value&...`

3.25.2. Notes

⚠ If you are using ActiveMQ

Note that the JMS component reuses Spring 2's `JmsTemplate` for sending messages. This is not ideal for use in a non-J2EE container and typically requires some caching in the JMS provider to avoid poor performance .

If you intend to use Apache ActiveMQ as your Message Broker, then we recommend that you either:

- Use the Section 3.1, "ActiveMQ" component, which is already configured to use ActiveMQ efficiently, or

- Use the `PoolingConnectionFactory` in ActiveMQ.

If you are consuming messages and using transactions (transacted=true) then the default settings for cache level can impact performance. If you are using XA transactions then you cannot cache as it can cause the XA transaction not to work properly. If you are not using XA, then you should consider caching as it speeds up performance, such as setting cacheLevelName=`CACHE_CONSUMER`.

The default setting for cacheLevelName is `CACHE_AUTO`. This default auto detects the mode and sets the cache level accordingly to: `CACHE_CONSUMER` if transacted is false, or `CACHE_NONE` if transacted is true. So you can say the default setting is conservative. Consider using cacheLevelName=`CACHE_CONSUMER` if you are using non-XA transactions.

If you wish to use durable topic subscriptions, you need to specify both **clientId** and **durableSubscriptionName**. The value of the `clientId` must be unique and can only be used by a single JMS connection instance in your entire network. You may prefer to use Virtual Topics instead to avoid this limitation. More background on durable messaging is available on the ActiveMQ site.

When using message headers, the JMS specification states that header names must be valid Java identifiers. So, by default, Camel ignores any headers that do not match this rule. So try to name your headers as if they are valid Java identifiers. One benefit of doing this is that you can then use your headers inside a JMS Selector (whose SQL92 syntax mandates Java identifier syntax for headers).

A simple strategy for mapping header names is used by default. The strategy is to replace any dots in the header name with the underscore character and to reverse the replacement when the header name is restored from a JMS message sent over the wire. What does this mean? No more losing method names to invoke on a bean component, no more losing the filename header for the File Component, and so on.

The current header name strategy for accepting header names in Camel is as follows:

- Dots are replaced by `_DOT_` and the replacement is reversed when Camel consumes the message. (for example, `org.apache.camel.MethodName` becomes `org_DOT_apache_DOT_camel_DOT_MethodName`).

- Hyphen is replaced by `_HYPHEN_` and the replacement is reversed when Camel consumes the message.

- Test if the name is a valid java identifier using the JDK core classes.

- If the test success, the header is added and sent over the wire; otherwise it is dropped (and logged at `DEBUG` level).

Are you using transactions?

If you are consuming messages, and have transacted=true, then the default settings for cache level can impact performance. Unfortunately the default setting is always CACHE_CONSUMER. However when you use transaction the cache level should be set to CACHE_NONE. This has been corrected in Camel 2.8 onwards where it auto detects the mode and set the level appropriately (the level is CACHE_AUTO). So the workaround is to set cacheLevelName=CACHE_NONE if you are using transactions.

3.25.3. Options

You can configure many different properties on the JMS endpoint which map to properties on the JMSConfiguration POJO. **Note:** Many of these properties map to properties on Spring JMS, which Camel uses for sending and receiving messages. You can get more information about these properties by consulting the relevant Spring documentation.

The options is divided into two tables, the first one with the most common options used. The latter contains the rest.

3.25.3.1. Most commonly used options

Option	Default Value	Description
clientId	null	Sets the JMS client ID to use. Note that this value, if specified, must be unique and can only be used by a single JMS connection instance. It is typically only required for durable topic subscriptions. You may prefer to use Virtual Topics instead.
concurrentConsumers	1	Specifies the default number of concurrent consumers.
disableReplyTo	false	If `true`, a producer will behave like a InOnly exchange with the exception that `JMSReplyTo` header is sent out and not be suppressed like in the case of `InOnly`. Like `InOnly` the producer will not wait for a reply. A consumer with this flag will behave like `InOnly`. This feature can be used to bridge `InOut` requests to another queue so that a route on the other queue will send it´s response directly back to the original `JMSReplyTo`.
durableSubscriptionName	null	The durable subscriber name for specifying durable topic subscriptions. The `clientId` option **must** be configured as well.
maxConcurrentConsumers	1	Specifies the maximum number of concurrent consumers.

Option	Default Value	Description
preserveMessageQos	false	Set to `true`, if you want to send message using the QoS settings specified on the message, instead of the QoS settings on the JMS endpoint. The following three headers are considered `JMSPriority`, `JMSDeliveryMode`, and `JMSExpiration`. You can provide all or only some of them. If not provided, Camel will fall back to use the values from the endpoint instead. So, when using this option, the headers override the values from the endpoint. The `explicitQosEnabled` option, by contrast, will only use options set on the endpoint, and not values from the message header.
replyTo	null	Provides an explicit ReplyTo destination, which overrides any incoming value of `Message.getJMSReplyTo()`. If you do [Request Reply] over JMS then read the section further below for more details.
replyToType	null	Allows to explicit specify which kind of strategy to use for replyTo queues when doing request/reply over JMS. Possible values are: {{Temporary}}, {{Shared}}, or {{Exclusive}}. By default Camel will use temporary queues. However if {{replyTo}} has been configured, then {{Shared}} is used by default. This option allows you to use exclusive instead of shared queues. Check the Camel website for more about this option.
requestTimeout	20000	(Producer only) The timeout for waiting for a reply when using the InOut Exchange Pattern (in milliseconds). See also the requestTimeoutCheckerInterval option.
selector	null	Sets the JMS Selector, which is an SQL 92 predicate that is used to filter messages within the broker. You may have to encode special characters such as = as %3D.
timeToLive	null	When sending messages, specifies the time-to-live of the message (in milliseconds).
transacted	false	Specifies whether to use transacted mode for sending/receiving messages using the InOnly Exchange Pattern.
testConnectionOnStartup	false	Specifies whether to test the connection on startup. This ensures that when Camel starts that all JMS consumers and producers have a valid connection to the JMS broker. If a connection cannot be granted then Camel throws an exception on startup. This ensures that Camel is not started with failed connections.

All the other options

Option	Default Value	Description
acceptMessagesWhile-Stopping	false	Specifies whether the consumer accept messages while it is stopping. You may consider enabling

Option	Default Value	Description
		this option, if you start and stop JMS routes at runtime, while there are still messages enqued on the queue. If this option is `false`, and you stop the `JMS` route, then messages may be rejected, and the JMS broker would have to attempt redeliveries, which yet again may be rejected, and eventually the message may be moved at a dead letter queue on the JMS broker. To avoid this its recommended to enable this option.
`acknowledgementModeName`	`AUTO_ACKNOWLEDGE`	The JMS acknowledgement name, which is one of: `TRANSACTED`, `CLIENT_ACKNOWLEDGE`, `AUTO_ACKNOWLEDGE`, `DUPS_OK_ACKNOWLEDGE`
`acknowledgementMode`	`-1`	The JMS acknowledgement mode defined as an Integer. Allows you to set vendor-specific extensions to the acknowledgment mode. For the regular modes, it is preferable to use the `acknowledgementModeName` instead.
`alwaysCopyMessage`	`false`	If `true`, Camel will always make a JMS message copy of the message when it is passed to the producer for sending. Copying the message is needed in some situations, such as when a `replyToDestinationSelectorName` is set (incidentally, Camel will set the `alwaysCopyMessage` option to `true`, if a `replyToDestinationSelectorName` is set)
`asyncStartListener`	`false`	Whether to startup the JmsConsumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to `true`, you will let routes startup, while the `JmsConsumer` connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages; You can then restart the route to retry.
`autoStartup`	`true`	Specifies whether the consumer container should auto-startup.
`asyncConsumer`	`false`	Whether the JmsConsumer processes the Exchange asynchronously using the Asynchronous Routing Engine. If enabled then the JmsConsumer may pick up the next message from the JMS queue, while the previous message is being processed asynchronously. This means that messages may be processed not 100% strictly in order. If disabled (as default) then the Exchange is fully processed before the JmsConsumer will pickup the next message from the JMS queue. Note if transactions have been

Option	Default Value	Description
		enabled, then asyncConsumer=true does not run asynchronously, as transactions must be executed synchronously.
cacheLevelName	CACHE_CONSUMER	Sets the cache level by name for the underlying JMS resources. Possible values are: CACHE_AUTO, CACHE_CONNECTION, CACHE_CONSUMER, CACHE_NONE, and CACHE_SESSION. See the Spring documentation and see the warning above.
cacheLevel	-1	Sets the cache level by ID for the underlying JMS resources.
consumerType	Default	The consumer type to use, which can be one of: Simple, Default or ServerSessionPool. The consumer type determines which Spring JMS listener to use. • Default will use org.springframework.jms.listener. DefaultMessageListenerContainer • Simple will use org.springframework.jms.listener. SimpleMessageListenerContainer • ServerSessionPool will use org.springframework.jms.listener. serversession.ServerSession- Message-ListenerContainer. • If the option, useVersion102=true, Camel will use the JMS 1.0.2 Spring classes instead.
connectionFactory	null	The default JMS connection factory to use for the listenerConnectionFactory and templateConnectionFactory, if neither is specified.
deliveryPersistent	true	Specifies whether persistent delivery is used by default.
destination	null	Specifies the JMS Destination object to use on this endpoint.
destinationName	null	Specifies the JMS destination name to use on this endpoint.
destinationResolver	null	A pluggable org.springframework.jms.support. destination.DestinationResolver that allows you to use your own resolver (for example, to lookup the real destination in a JNDI registry).
disableTimeToLive	false	Use this option to force disabling time to live. For example when you do request/reply over JMS, then Camel will by default use the {{requestTimeout}} value as time to live on the message being sent. The problem is that the sender and receiver systems have to have their

Option	Default Value	Description
		clocks synchronized, so they are in sync. This is not always so easy to archive. So you can use {{disableTimeToLive=true}} to *not* set a time to live value on the send message. Then the message will not expire on the receiver system.
`eagerLoadingOfProperties`	`false`	Enables eager loading of JMS properties as soon as a message is received, which is generally inefficient, because the JMS properties might not be required. However, this feature can sometimes catch any issues with the underlying JMS provider and the use of JMS properties at an early stage. This feature can also be used for testing purposes, to ensure JMS properties can be understood and handled correctly.
`exceptionListener`	`null`	Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.
`errorHandler`	`null`	Specifies a org.springframework.util.ErrorHandler to be invoked in case of any uncaught exceptions thrown while processing a message. By default these exceptions will be logged at the WARN level, if no errorHandler has been configured. From Camel 2.9.1 onwards you can configure logging level and whether stack traces should be logged using the below two options. This makes it much easier to configure, than having to code a custom errorHandler.
`errorHandlerLoggingLevel`	`WARN`	Allows for configuring the default errorHandler logging level for logging uncaught exceptions.
`errorHandlerLogStackTrace`	`true`	Allows to control whether stacktraces should be logged or not, by the default errorHandler.
`explicitQosEnabled`	`false`	Set if the `deliveryMode`, `priority` or `timeToLive` qualities of service should be used when sending messages. This option is based on Spring's `JmsTemplate`. The `deliveryMode`, `priority` and `timeToLive` options are applied to the current endpoint. This contrasts with the `preserveMessageQos` option, which operates at message granularity, reading QoS properties exclusively from the Camel In message headers.
`exposeListenerSession`	`true`	Specifies whether the listener session should be exposed when consuming messages.
`forceSendOriginalMessage`	`false`	When using mapJmsMessage=false Camel will create a new JMS message to send to a new JMS destination if you touch the headers (get or set) during the route. Set this option to `true` to force Camel to send the original JMS message that was received.
`idleConsumerLimit`	`1`	Specify the limit for the number of consumers that are allowed to be idle at any given time.
`idleTaskExecutionLimit`	`1`	Specifies the limit for idle executions of a receive task, not having received any message within its

Option	Default Value	Description
		execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in the case of dynamic scheduling; see the `maxConcurrentConsumers` setting).
`jmsMessageType`	`null`	Allows you to force the use of a specific `javax.jms.Message` implementation for sending JMS messages. Possible values are: `Bytes`, `Map`, `Object`, `Stream`, `Text`. By default, Camel would determine which JMS message type to use from the In body type. This option allows you to specify it.
`jmsKeyFormatStrategy`	`default`	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides two implementations out of the box: `default` and `passthrough`. The `default` strategy will safely marshal dots ('.') and hyphens ('-') The `passthrough` strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the `org.apache.camel.component.jms.JmsKeyFormatStrategy` and refer to it using the # notation.
`jmsOperations`	`null`	Allows you to use your own implementation of the `org.springframework.jms.core.JmsOperations` interface. Camel uses `JmsTemplate` as default. Can be used for testing purpose (rarely used, as stated in the Spring API docs) .
`lazyCreateTransaction-Manager`	`true`	If `true`, Camel will create a `JmsTransactionManager`, if there is no `transactionManager` injected when option `transacted=true`.
`listenerConnection-Factory`	`null`	The JMS connection factory used for consuming messages.
`mapJmsMessage`	`true`	Specifies whether Camel should auto map the received JMS message to an appropiate payload type, such as `javax.jms.TextMessage` to a `String` etc. See section about how mapping works below for more details.
`maxMessagesPerTask`	`-1`	The number of messages per task. -1 is unlimited.
`maximumBrowseSize`	`-1`	Limits the number of messages fetched at most, when browsing endpoints using Browse or JMX API.
`messageConverter`	`null`	To use a custom Spring `org.springframework.jms.support.converter.MessageConverter` so you can be totally in control how to map to and from a `javax.jms.Message`.

Option	Default Value	Description
`messageIdEnabled`	`true`	When sending, specifies whether message IDs should be added.
`messageTimestampEnabled`	`true`	Specifies whether timestamps should be enabled by default on sending messages.
`password`	`null`	The password for the connector factory.
`priority`	4	Values greater than 1 specify the message priority when sending (where 0 is the lowest priority and 9 is the highest). The `explicitQosEnabled` option **must** also be enabled in order for this option to have any effect.
`pubSubNoLocal`	`false`	Specifies whether to inhibit the delivery of messages published by its own connection.
`receiveTimeout`	*None*	The timeout for receiving messages (in milliseconds).
`recoveryInterval`	5000	Specifies the interval between recovery attempts, that is, when a connection is being refreshed, in milliseconds. The default is 5000 ms, that is, 5 seconds.
`replyManager-ExecutorService`		Allows you to configure an explicit ScheduledExecutorService to be used for processing timed out exchanges when doing request/reply over JMS. By default a single threaded pool is used. If you expect many timeouts when doing request/reply you can process the timedout exchanges using concurrency by configuring a thread pool which has a higher number of threads. See Threading Model for how to configure thread pools. See also the `requestTimeout` which is the timeout value for request/reply.
`replyToCacheLevelName`	–	Sets the cache level by name for the reply consumer when doing request/reply over JMS. This option only applies when using fixed reply queues (not temporary). Camel will by default use: `CACHE_CONSUMER` for exclusive or shared w/ `{{replyToSelectorName}}`. And `CACHE_SESSION` for shared without `replyToSelectorName`. Some JMS brokers such as IBM WebSphere may require to set the replyToCacheLevelName=`CACHE_NONE` to work.
`replyToDestination-SelectorName`	`null`	Sets the JMS Selector using the fixed name to be used so you can filter out your own replies from the others when using a shared queue (that is, if you are not using a temporary reply queue).
`replyToDelivery-Persistent`	`true`	Specifies whether to use persistent delivery by default for replies.
`requestTimeout-CheckerInterval`	1000	Configures how often Camel should check for timed out Exchanges when doing request/reply over JMS. By default Camel checks once per second. But if you must react faster when a timeout occurs, then you can lower this

Option	Default Value	Description
		interval, to check more frequently. The timeout is determined by the requestTimeout option.
subscriptionDurable	false	**@deprecated:** Enabled by default, if you specify a durableSubscriberName and a clientId.
taskExecutor	null	Allows you to specify a custom task executor for consuming messages.
taskExecutorSpring2	null	**Camel 2.6:** To use when using Spring 2.x with Camel. Allows you to specify a custom task executor for consuming messages.
templateConnection-Factory	null	The JMS connection factory used for sending messages.
transactedInOut	false	**@deprecated:** Specifies whether to use transacted mode for sending messages using the InOut Exchange Pattern. Applies only to producer endpoints. See section Enabling Transacted Consumption for more details.
transactionManager	null	The Spring transaction manager to use.
transactionName	JmsConsumer [destinationName]	The name of the transaction to use.
transactionTimeout	null	The timeout value of the transaction, if using transacted mode.
transferException	false	If enabled and you are using Section 2.39, "Request Reply" messaging (InOut) and an Exchange failed on the consumer side, then the caused Exception will be send back in response as a javax.jms.ObjectMessage. If the client is Camel, the returned Exception is rethrown. This allows you to use Camel Section 3.25, "JMS" as a bridge in your routing; for example, using persistent queues to enable robust routing. Notice that if you also have **transferExchange** enabled, this option takes precedence. The caught exception is required to be serializable. The original Exception on the consumer side can be wrapped in an outer exception such as org.apache.camel.RuntimeCamelException when returned to the producer.
transferExchange	false	You can transfer the exchange over the wire instead of just the body and headers. The following fields are transferred: In body, Out body, Fault body, In headers, Out headers, Fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level.
username	null	The username for the connector factory.
useMessageIDAs-CorrelationID	false	Specifies whether JMSMessageID should always be used as JMSCorrelationID for **InOut** messages.

Message Mapping between JMS and CamelCamel automatically maps messages between `javax.jms.Message` and `org.apache.camel.Message`. When sending a JMS message, Camel converts the message body to the following JMS message types:

Body Type	JMS Message	Comment
`String`	`javax.jms.TextMessage`	
`org.w3c.dom.Node`	`javax.jms.TextMessage`	The DOM will be converted to `String`.
`Map`	`javax.jms.MapMessage`	
`java.io.Serializable`	`javax.jms.ObjectMessage`	
`byte[]`	`javax.jms.BytesMessage`	
`java.io.File`	`javax.jms.BytesMessage`	
`java.io.Reader`	`javax.jms.BytesMessage`	
`java.io.InputStream`	`javax.jms.BytesMessage`	
`java.nio.ByteBuffer`	`javax.jms.BytesMessage`	

When receiving a JMS message, Camel converts the JMS message to the following body type:

JMS Message	Body Type
`javax.jms.TextMessage`	`String`
`javax.jms.BytesMessage`	`byte[]`
`javax.jms.MapMessage`	`Map<String, Object>`
`javax.jms.ObjectMessage`	`Object`

3.25.4. Message format when sending

The exchange that is sent over the JMS wire must conform to the JMS Message spec.

For the `exchange.in.header` the following rules apply for the header **keys** :

- Keys starting with `JMS` or `JMSX` are reserved.

- `exchange.in.headers` keys must be literals and all be valid Java identifiers (do not use dots in the key name).

- Camel replaces dots and hyphens with underscores in key names ('.' is replaced by `_DOT_` and '_' is replaced by `_HYPHEN_`). This replacement is reversed when Camel consumes JMS messages.

- See also the option `jmsKeyFormatStrategy`, which allows you to use your own custom strategy for formatting keys.

For the `exchange.in.header`, the following rules apply for the header **values** :

- The values must be primitives or their counter objects (such as `Integer`, `Long`, `Character`). The types, `String`, `CharSequence`, `Date`, `BigDecimal` and `BigInteger` are all converted to their `toString()` representation. All other types are dropped.

Camel will log with category `org.apache.camel.component.jms.JmsBinding` at **DEBUG** level if it drops a given header value. For example:

```
2008-07-09 06:43:04,046 [main            ] DEBUG JmsBinding
- Ignoring non primitive header: order of class: org.apache.camel.component
.jms.issues.DummyOrder with value: DummyOrder{orderId=333, itemId=4444,
quantity=2}
```

3.25.5. Message format when receiving

Camel adds the following properties to the `Exchange` when it receives a message:

Property	Type	Description
`org.apache.camel.jms.` `replyDestination`	`javax.jms.Destination`	The reply destination.

Camel adds the following JMS properties to the In message headers when it receives a JMS message:

Header	Type	Description
`JMSCorrelationID`	`String`	The JMS correlation ID.
`JMSDeliveryMode`	`int`	The JMS delivery mode.
`JMSDestination`	`javax.jms.Destination`	The JMS destination.
`JMSExpiration`	`long`	The JMS expiration.
`JMSMessageID`	`String`	The JMS unique message ID.
`JMSPriority`	`int`	The JMS priority (with 0 as the lowest priority and 9 as the highest).
`JMSRedelivered`	`boolean`	the JMS message redelivered.
`JMSReplyTo`	`javax.jms.Destination`	The JMS reply-to destination.
`JMSTimestamp`	`long`	The JMS timestamp.
`JMSType`	`String`	The JMS type.
`JMSXGroupID`	`String`	The JMS group ID.

As all the above information is standard JMS you can check the JMS documentation for further details.

Using Camel JMS to send and receive messages

The JMS component is complex and you have to pay close attention to how it works in some cases. So this is a short summary of some of the areas/pitfalls to look for.

When Camel sends a message using its `JMSProducer`, it checks the following conditions:

- The message exchange pattern,

- Whether a `JMSReplyTo` was set in the endpoint or in the message headers,

- Whether any of the following options have been set on the JMS endpoint: `disableReplyTo`, `preserveMessageQos`, `explicitQosEnabled`.

All this can be complex to understand and configure to support your use case.

3.25.6.1. JmsProducer

The `JmsProducer` behaves as follows, depending on configuration:

Exchange Pattern	Other options	Description
InOut	-	Camel will expect a reply, set a temporary `JMSReplyTo`, and after sending the message, it will start to listen for the reply message on the temporary queue.

Exchange Pattern	Other options	Description
InOut	JMSReplyTo is set	Camel will expect a reply and, after sending the message, it will start to listen for the reply message on the specified JMSReplyTo queue.
InOnly	-	Camel will send the message and **not** expect a reply.
InOnly	JMSReplyTo is set	By default, Camel discards the JMSReplyTo destination and clears the JMSReplyTo header before sending the message. Camel then sends the message and does **not** expect a reply. Camel logs this in the log at DEBUG level. You can use preserveMessageQuo=true to instruct Camel to keep the JMSReplyTo. In all situations the JmsProducer does **not** expect any reply and thus continue after sending the message.

3.25.6.2. JmsConsumer

The JmsConsumer behaves as follows, depending on configuration:

Exchange Pattern	Other options	Description
InOut	-	Camel will send the reply back to the JMSReplyTo queue.
InOnly	-	Camel will not send a reply back, as the pattern is *InOnly*.
-	disableReplyTo=true	This option suppresses replies.

Thus, pay attention to the message exchange pattern set on your exchanges.

If you send a message to a JMS destination in the middle of your route you can specify the exchange pattern to use, see more at Section 2.39, "Request Reply". This is useful if you want to send an InOnly message to a JMS topic:

```
from("activemq:queue:in")
   .to("bean:validateOrder")
   .to(ExchangePattern.InOnly, "activemq:topic:order")
   .to("bean:handleOrder");
```

3.25.7. Configuring different JMS providers

You can configure your JMS provider in Spring XML as follows:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <jmxAgent id="agent" disabled="true"/>
</camelContext>

<bean id="activemq"
      class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value=
        "vm://localhost?broker.persistent=false&broker.useJmx=false"/>
    </bean>
```

```
    </property>
</bean>
```

Basically, you can configure as many JMS component instances as you wish and give them **a unique name using the id attribute** . The preceding example configures an `activemq` component. You could do the same to configure MQSeries, TibCo, BEA, Sonic and so on.

Once you have a named JMS component, you can then refer to endpoints within that component using URIs. For example for the component name, `activemq`, you can then refer to destinations using the URI format, `activemq:[queue:|topic:]destinationName`. You can use the same approach for all other JMS providers.

This works by the SpringCamelContext lazily fetching components from the Spring context for the scheme name you use for Endpoint URIs and having the Component resolve the endpoint URIs.

3.25.8. Samples

JMS is used in many examples for other components as well. But we provide a few samples below to get started.

3.25.8.1. Receiving from JMS

In the following sample we configure a route that receives JMS messages and routes the message to a POJO:

```
from("jms:queue:foo").
    to("bean:myBusinessLogic");
```

You can of course use any of the EIP patterns so the route can be context based. For example, here's how to filter an order topic for the big spenders:

```
from("jms:topic:OrdersTopic").
  filter().method("myBean", "isGoldCustomer").
    to("jms:queue:BigSpendersQueue");
```

3.25.8.2. Sending to a JMS

In the sample below we poll a file folder and send the file content to a JMS topic. As we want the content of the file as a `TextMessage` instead of a `BytesMessage`, we need to convert the body to a `String` :

```
from("file://orders").
  convertBodyTo(String.class).
  to("jms:topic:OrdersTopic");
```

3.25.8.3. Using Annotations

Camel also has annotations so you can use POJO Consuming and POJO Producing.

3.25.8.4. Spring DSL sample

The preceding examples use the Java DSL. Camel also supports Spring XML DSL. Here is the big spender sample using Spring DSL:

```
<route>
  <from uri="jms:topic:OrdersTopic"/>
  <filter>
    <method bean="myBean" method="isGoldCustomer"/>
    <to uri="jms:queue:BigSpendersQueue"/>
  </filter>
</route>
```

3.25.8.5. Other samples

JMS appears in many of the examples for other components and EIP patterns, as well in the online Apache Camel documentation. A recommended tutorial is this one that uses JMS but focuses on how well Spring Remoting and Camel work together Tutorial-JmsRemoting.

3.26. JMX

Available as of Camel 2.6

Component allows consumers to subscribe to an mbean's Notifications. The component supports passing the Notification object directly through the Exchange or serializing it to XML according to the schema provided within this project. This is a consumer only component. Exceptions are thrown if you attempt to create a producer for it.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jmx</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.26.1. URI Format

The component can connect to the local platform mbean server with the following URI:

```
jmx://platform?options
```

A remote mbean server url can be provided following the initial JMX scheme like so:

```
jmx:service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi?options
```

You can append query options to the URI in the following format, ?options=value&option2=value&...

3.26.2. URI Options

Property	Required	Default	Description
format	no	xml	Format for the message body. Either "xml" or "raw". If xml, the notification is serialized to xml. If raw, then the raw java object is set as the body.

Property	Required	Default	Description
user	no		Credentials for making a remote connection.
password	no		Credentials for making a remote connection.
objectDomain	yes		The domain for the mbean you're connecting to.
objectName	no		The name key for the mbean you're connecting to. This value is mutually exclusive with the object properties that get passed. (see below)
notificationFilter	no		Reference to a bean that implements the `NotificationFilter`. The #ref syntax should be used to reference the bean via the Registry.
handback	no		Value to handback to the listener when a notification is received. This value will be put in the message header with the key "jmx.handback"

3.26.3. ObjectName Construction

The URI must always have the objectDomain property. In addition, the URI must contain either objectName or one or more properties that start with "key."

3.26.4. Domain with Name property

When the objectName property is provided, the following constructor is used to build the ObjectName? for the mbean:

```
ObjectName(String domain, String key, String value)
```

The key value in the above will be "name" and the value will be the value of the objectName property.

3.26.5. Domain with Hashtable

```
ObjectName(String domain, Hashtable<String,String> table)
```

The Hashtable is constructed by extracting properties that start with "key." The properties will have the "key." prefixed stripped prior to building the Hashtable. This allows the URI to contain a variable number of properties to identify the mbean.

3.26.6. Example

```
from("jmx:platform?objectDomain=jmxExample&key.name=simpleBean").
        to("log:jmxEvent");
```

A full example is here.

3.27. JPA

The **jpa** component enables you to store and retrieve Java objects from persistent storage using EJB 3's Java Persistence Architecture (JPA), which is a standard interface layer that wraps Object/Relational Mapping (ORM) products such as OpenJPA, Hibernate, TopLink, and so on.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jpa</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.27.1. Sending to the endpoint

You can store a Java entity bean in a database by sending it to a JPA producer endpoint. The body of the *In* message is assumed to be an entity bean (that is, a POJO with an @Entity annotation on it) or a collection or array of entity beans.

If the body does not contain one of the previous listed types, put a Section 2.29, "Message Translator" in front of the endpoint to perform the necessary conversion first.

3.27.2. Consuming from the endpoint

Consuming messages from a JPA consumer endpoint removes (or updates) entity beans in the database. This allows you to use a database table as a logical queue: consumers take messages from the queue and then delete/ update them to logically remove them from the queue.

If you do not wish to delete the entity bean when it has been processed, you can specify `consumeDelete=false` on the URI. This will result in the entity being processed each poll.

If you would rather perform some update on the entity to mark it as processed (such as to exclude it from a future query) then you can annotate a method with @Consumed which will be invoked on your entity bean when the entity bean is consumed.

3.27.3. URI format

```
jpa:[entityClassName][?options]
```

For sending to the endpoint, the *entityClassName* is optional. If specified, it helps the Type Converter to ensure the body is of the correct type.

For consuming, the *entityClassName* is mandatory.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.27.4. Options

Name	Default Value	Description
entityType	*entityClassName*	Overrides the *entityClassName* from the URI.
persistenceUnit	camel	The JPA persistence unit used by default.
consumeDelete	true	**JPA consumer only:** If true, the entity is deleted after it is consumed; if false, the entity is not deleted.
consumeLockEntity	true	**JPA consumer only:** Specifies whether or not to set an exclusive lock on each entity bean while processing the results from polling.
flushOnSend	true	**JPA producer only:** Flushes the EntityManager after the entity bean has been persisted.
maximumResults	-1	**JPA consumer only:** Set the maximum number of results to retrieve on the Query.
transactionManager	null	Specifies the transaction manager to use. If none provided, Camel will use a JpaTransactionManager by default. Can be used to set a JTA transaction manager (for integration with an EJB container).
consumer.delay	500	**JPA consumer only:** Delay in milliseconds between each poll.
consumer.initialDelay	1000	**JPA consumer only:** Milliseconds before polling starts.
consumer.useFixedDelay	false	**JPA consumer only:** Set to true to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
maxMessagesPerPoll	0	**JPA consumer only:** An integer value to define the maximum number of messages to gather per poll. By default, no maximum is set. Can be used to avoid polling many thousands of messages when starting up the server. Set a value of 0 or negative to disable.
consumer.query		**JPA consumer only:** To use a custom query when consuming data.
consumer.namedQuery		**JPA consumer only:** To use a named query when consuming data.
consumer.nativeQuery		**JPA consumer only:** To use a custom native query when consuming data.
consumer.resultClass		**Camel 2.7: JPA consumer only:** Defines the type of the returned payload (we will call entityManager.createNativeQuery (nativeQuery, resultClass) instead of entityManager.createNativeQuery (nativeQuery)). Without this option, we will return an object array. Only has an affect when using in conjunction with native query when consuming data.
usePersist	false	**JPA producer only:** Indicates to use entityManager.persist(entity)

Name	Default Value	Description
		instead of `entityManager.merge(entity)`. Note: `entityManager.persist(entity)` doesn't work for detached entities (where the EntityManager has to execute an UPDATE instead of an INSERT query)!

3.27.5. Message Headers

Camel adds the following message headers to the exchange:

Header	Type	Description
`CamelJpaTemplate`	`JpaTemplate`	The `JpaTemplate` object that is used to access the entity bean. You need this object in some situations, for instance in a type converter or when you are doing some custom processing.

3.27.6. Configuring EntityManagerFactory

It is strongly advised to configure the JPA component to use a specific `EntityManagerFactory` instance. If failed to do so each `JpaEndpoint` will auto create their own instance of `EntityManagerFactory` which most often is not what you want.

For example, you can instantiate a JPA component that references the `myEMFactory` entity manager factory, as follows:

```
<bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent">
   <property name="entityManagerFactory" ref="myEMFactory"/>
</bean>
```

In **Camel 2.3** the `JpaComponent` will auto lookup the `EntityManagerFactory` from the Registry which means you do not need to configure this on the `JpaComponent` as shown above. You only need to do so if there is ambiguity, in which case Camel will log a WARN.

3.27.7. Configuring TransactionManager

It is strongly advised to configure the `TransactionManager` instance used by the JPA component. If failed to do so each `JpaEndpoint` will auto create their own instance of `TransactionManager` which most often is not what you want.

For example, you can instantiate a JPA component that references the `myTransactionManager` transaction manager, as follows:

```
<bean id="jpa" class="org.apache.camel.component.jpa.JpaComponent">
   <property name="entityManagerFactory" ref="myEMFactory"/>
   <property name="transactionManager" ref="myTransactionManager"/>
</bean>
```

In **Camel 2.3** the `JpaComponent` will auto lookup the `TransactionManager` from the Registry which means you do not need to configure this on the `JpaComponent` as shown above. You only need to do so if there is ambiguity, in which case Camel will log a WARN.

3.27.8. Using a consumer with a named query

For consuming only selected entities, you can use the `consumer.namedQuery` URI query option. First, you have to define the named query in the JPA Entity class:

```
@Entity
@NamedQuery(name = "step1",
    query = "select x from MultiSteps x where x.step = 1")
public class MultiSteps {
    ...
}
```

After that you can define a consumer uri like this one:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.namedQuery=step1")
.to("bean:myBusinessLogic");
```

3.27.9. Using a consumer with a query

For consuming only selected entities, you can use the `consumer.query` URI query option. You only have to define the query option:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.query=
select o from org.apache.camel.examples.MultiSteps o where o.step = 1")
.to("bean:myBusinessLogic");
```

3.27.10. Using a consumer with a native query

For consuming only selected entities, you can use the `consumer.nativeQuery` URI query option. You only have to define the native query option:

```
from("jpa://org.apache.camel.examples.MultiSteps?consumer.nativeQuery=
select * from MultiSteps where step = 1")
.to("bean:myBusinessLogic");
```

If you use the native query option, you will receive an object array in the message body.

3.27.11. Example

See Tracer Example for an example using Section 3.27, "JPA" to store traced messages into a database.

3.28. Jsch

The camel-jsch component supports the SCP protocol using the Client API of the Jsch project. Jsch is already used in Camel by the FTP component for the sftp: protocol. Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jsch</artifactId>
```

```
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.28.1. URI format and options

```
scp://host[:port]/destination[?options]
```

You can append query options to the URI in the following format: ?option=value&option=value&...

The file name can be specified either in the <path> part of the URI or as a "CamelFileName" header on the message (Exchange.FILE_NAME if used in code).

Options

Name	Default	Description
username	null	Specifies the username to use to log in to the remote file system.
password	null	Specifies the password to use to log in to the remote file system.
knownHostsFile	null	Sets the known_hosts file, so that the scp endpoint can do host key verification.
strictHostKeyChecking	no	Sets whether to use strict host key checking. Possible values are: no, yes
chmod	null	Allows you to set chmod on the stored file. For example chmod=664.

3.28.2. Limitations

Currently camel-jsch only supports a Producer (i.e. copy files to another host). The reason is that the scp protocol does not offer the possibility to scan (list) the content of a directory. As such a polling consumer cannot watch for changes and trigger events on changes. It is possible however to use camel-jsch in sink mode for one time copy from a remote host using a ConsumerTemplate (see Polling Consumer for more details). If continuous monitoring of a directory on a remote host and secure transfer is required, you can consider using the sftp protocol.

3.29. Log

The **log:** component logs message exchanges to the underlying logging mechanism.

Camel uses commons-logging which allows you to configure logging via

• Log4j

• JDK 1.4 logging

• Avalon

• SimpleLog - a simple provider in commons-logging

Refer to the commons-logging user guide for a more complete overview of how to use and configure commons-logging.

3.29.1. URI format and Options

```
log:loggingCategory[?options]
```

where **loggingCategory** is the name of the logging category to use. You can append query options to the URI in the following format, `?option=value&option=value&...`

For example, a log endpoint typically specifies the logging level using the `level` option, as follows:

```
log:org.apache.camel.example?level=DEBUG
```

The default logger logs every exchange (*regular logging*). But Camel also ships with the `Throughput` logger, which is used whenever the `groupSize` option is specified.

Also a log in the DSL

There is also a `log` directly in the DSL, but it has a different purpose. It is meant for lightweight and human logs. See more details at Section 2.20, "Log".

Options

Option	Default	Type	Description
level	INFO	String	Logging level to use. Possible values: `FATAL, ERROR, WARN, INFO, DEBUG, TRACE, OFF`
marker	null	String	An optional Marker name to use.
groupSize	null	Integer	An integer that specifies a group size for throughput logging.
groupInterval	null	Integer	If specified will group message stats by this time interval (in milliseconds)
groupDelay	0	Integer	Set the initial delay for stats (in milliseconds)
groupActiveOnly	true	boolean	If true, will hide stats when no new messages have been received for a time interval, if false, show stats regardless of message traffic.

Note: groupDelay and groupActiveOnly are only applicable when using groupInterval

3.29.2. Formatting

The log formats the execution of exchanges to log lines. By default, the log uses `LogFormatter` to format the log output, where `LogFormatter` has the following options:

Option	Default	Description
showAll	false	Quick option for turning all options on. (multiline, maxChars has to be manually set if to be used)
showExchangeId	false	Show the unique exchange ID.
showExchangePattern	true	Shows the Message Exchange Pattern (or MEP for short).
showProperties	false	Show the exchange properties.
showHeaders	false	Show the In message headers.

Option	Default	Description
`showBodyType`	`true`	Show the In body Java type.
`showBody`	`true`	Show the In body.
`showOut`	`false`	If the exchange has an Out message, show the Out message.
`showException`	`false`	If the exchange has an exception, show the exception message (no stack trace).
`showCaughtException`	`false`	If the exchange has a caught exception, show the exception message (no stack trace). A caught exception is stored as a property on the exchange (using the key Exchange.EXCEPTION_CAUGHT) and for instance a `doCatch` can catch exceptions. See Try Catch Finally.
`showStackTrace`	`false`	Show the stack trace, if an exchange has an exception. Only effective if one of `showAll`, `showException` or `showCaughtException` are enabled.
`showFiles`	`false`	Whether Camel should show file bodies or not (eg such as java.io.File).
`showFuture`	`false`	Whether Camel should show `java.util.concurrent.Future` bodies or not. If enabled Camel could potentially wait until the `Future` task is done. By default, this will not wait.
`showStreams`	`false`	Whether Camel should show stream bodies or not (eg such as java.io.InputStream). Beware if you enable this option then you may not be able later to access the message body as the stream have already been read by this logger. To remedy this you will have to use Stream Caching.
`multiline`	`false`	If `true`, each piece of information is logged on a new line.
`maxChars`		Limits the number of characters logged per line.

3.29.3. Regular logger sample

In the route below we log the incoming orders at `DEBUG` level before the order is processed:

```
from("activemq:orders")
   .to("log:com.mycompany.order?level=DEBUG")
   .to("bean:processOrder");
```

Or using Spring XML to define the route:

```
<route>
   <from uri="activemq:orders"/>
   <to uri="log:com.mycompany.order?level=DEBUG"/>
   <to uri="bean:processOrder"/>
</route>
```

3.29.4. Regular logger with formatter sample

In the route below we log the incoming orders at `INFO` level before the order is processed.

```
from("activemq:orders")
```

```
    .to("log:com.mycompany.order?showAll=true&multiline=true")
    .to("bean:processOrder");
```

3.29.5. Throughput logger with groupSize sample

In the route below we log the throughput of the incoming orders at DEBUG level grouped by 10 messages.

```
from("activemq:orders")
    .to("log:com.mycompany.order?level=DEBUG?groupSize=10")
    .to("bean:processOrder");
```

3.29.6. Throughput logger with groupInterval sample

This route will result in message stats logged every 10s, with an initial 60s delay and stats displayed even if there isn't any message traffic.

```
from("activemq:orders")
      .to("log:com.mycompany.order?level=DEBUG?groupInterval=10000&group
Delay=60000&groupActiveOnly=false")
      .to("bean:processOrder");
```

The following will be logged:

```
"Received: 1000 new messages, with total 2000 so far. Last group took:
10000 millis which is: 100 messages per second. average: 100"
```

3.30. Lucene

The **lucene** component is based on the Apache Lucene project. Apache Lucene is a powerful high-performance, full-featured text search engine library written entirely in Java. For more details about Lucene, please see the following links

• http://lucene.apache.org/java/docs/

• http://lucene.apache.org/java/docs/features.html

The lucene component in Camel facilitates integration and utilization of Lucene endpoints in enterprise integration patterns and scenarios. The lucene component does the following

• builds a searchable index of documents when payloads are sent to the Lucene Endpoint

• facilitates performing of indexed searches in Camel

This component only supports producer endpoints.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-lucene</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.30.1. URI format

```
lucene:searcherName:insert[?options]
lucene:searcherName:query[?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.30.2. Insert Options

Name	Default Value	Description
analyzer	StandardAnalyzer	An Analyzer builds TokenStreams, which analyze text. It thus represents a policy for extracting index terms from text. The value for analyzer can be any class that extends the abstract class org.apache.lucene.analysis.Analyzer. Lucene also offers a rich set of analyzers out of the box
indexDir	./indexDirectory	A file system directory in which index files are created upon analysis of the document by the specified analyzer
srcDir	null	An optional directory containing files to be used to be analyzed and added to the index at producer startup.

3.30.3. Query Options

Name	Default Value	Description
analyzer	StandardAnalyzer	An Analyzer builds TokenStreams, which analyze text. It thus represents a policy for extracting index terms from text. The value for analyzer can be any class that extends the abstract class org.apache.lucene.analysis.Analyzer. Lucene also offers a rich set of analyzers out of the box
indexDir	./indexDirectory	A file system directory in which index files are created upon analysis of the document by the specified analyzer
maxHits	10	An integer value that limits the result set of the search operation

3.30.4. Sending/Receiving Messages to/from the cache

3.30.4.1. Message Headers

Header	Description
QUERY	The Lucene Query to performed on the index. The query may include wildcards and phrases

3.30.4.2. Lucene Producers

This component supports two producer endpoints.

- **insert**: the insert producer builds a searchable index by analyzing the body in incoming exchanges and associating it with a token ("content").

- **query**: the query producer performs searches on a pre-created index. The query uses the searchable index to perform score & relevance based searches. Queries are sent via the incoming exchange contains a header property name called 'QUERY'. The value of the header property 'QUERY' is a Lucene Query. For more details on how to create Lucene Queries check out http://lucene.apache.org/java/3_0_0/queryparsersyntax.html

3.30.4.3. Lucene Processor

There is a processor called LuceneQueryProcessor available to perform queries against lucene without the need to create a producer.

3.30.5. Lucene Usage Samples

3.30.5.1. Example 1: Creating a Lucene index

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
      from("direct:start")
        .to("lucene:whitespaceQuotesIndex:insert?analyzer=
           #whitespaceAnalyzer&indexDir=#whitespace&srcDir=#load_dir")
        .to("mock:result");
   }
};
```

3.30.5.2. Example 2: Loading properties into the JNDI registry in the Camel Context

```
@Override
protected JndiRegistry createRegistry() throws Exception {
   JndiRegistry registry = new JndiRegistry(createJndiContext());
   registry.bind("whitespace", new File("./whitespaceIndexDir"));
   registry.bind("load_dir", new File("src/test/resources/sources"));
   registry.bind("whitespaceAnalyzer", new WhitespaceAnalyzer());
   return registry;
}
...
CamelContext context = new DefaultCamelContext(createRegistry());
```

3.30.5.3. Example 2: Performing searches using a Query Producer

```
RouteBuilder builder = new RouteBuilder() {
```

```
    public void configure() {
        from("direct:start").
          setHeader("QUERY", constant("Seinfeld")).
          to("lucene:searchIndex:query?
            analyzer=#whitespaceAnalyzer&indexDir=#whitespace&maxHits=20").
          to("direct:next");

        from("direct:next").process(new Processor() {
          public void process(Exchange exchange) throws Exception {
            Hits hits = exchange.getIn().getBody(Hits.class);
            printResults(hits);
          }

          private void printResults(Hits hits) {
            LOG.debug("Number of hits: " + hits.getNumberOfHits());
            for (int i = 0; i < hits.getNumberOfHits(); i++) {
              LOG.debug("Hit " + i + " Index Location:"
                + hits.getHit().get(i).getHitLocation());
              LOG.debug("Hit " + i + " Score:"
                + hits.getHit().get(i).getScore());
              LOG.debug("Hit " + i + " Data:"
                + hits.getHit().get(i).getData());
            }
          }
        }).to("mock:searchResult");
    }
};
```

3.30.5.4. Example 3: Performing searches using a Query Processor

```
RouteBuilder builder = new RouteBuilder() {
   public void configure() {
       try {
           from("direct:start").
               setHeader("QUERY", constant("Rodney Dangerfield")).
               process(new LuceneQueryProcessor(
                   "target/stdindexDir", analyzer, null, 20)).
               to("direct:next");
       } catch (Exception e) {
           e.printStackTrace();
       }

       from("direct:next").process(new Processor() {
           public void process(Exchange exchange) throws Exception {
               Hits hits = exchange.getIn().getBody(Hits.class);
               printResults(hits);
           }

           private void printResults(Hits hits) {
               LOG.debug("Number of hits: " + hits.getNumberOfHits());
               for (int i = 0; i < hits.getNumberOfHits(); i++) {
                   LOG.debug("Hit " + i + " Index Location:" +
                       hits.getHit().get(i).getHitLocation());
                   LOG.debug("Hit " + i + " Score:" +
```

```
                    hits.getHit().get(i).getScore());
              LOG.debug("Hit " + i + " Data:" +
                    hits.getHit().get(i).getData());
          }
        }
      }).to("mock:searchResult");
    }
};
```

3.31. Mail

The mail component provides access to Email via Spring's Mail support and the underlying JavaMail system.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-mail</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.31.1. URI format

Mail endpoints can have one of the following URI formats (for the protocols, SMTP, POP3, or IMAP, respectively):

```
smtp://[username@]host[:port][?options]
pop3://[username@]host[:port][?options]
imap://[username@]host[:port][?options]
```

The mail component also supports secure variants of these protocols (layered over SSL). You can enable the secure protocols by adding `s` to the scheme:

```
smtps://[username@]host[:port][?options]
pop3s://[username@]host[:port][?options]
imaps://[username@]host[:port][?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.31.1.1. Sample endpoints

Typically, you specify a URI with login credentials as follows (taking SMTP as an example):

```
smtp://[username@]host[:port][?password=somepwd]
```

Alternatively, it is possible to specify both the user name and the password as query options:

```
smtp://host[:port]?password=somepwd&username=someuser
```

For example:

```
smtp://mycompany.mailserver:30?password=tiger&username=scott
```

3.31.1.2. Default ports

Default port numbers are supported. If the port number is omitted, Camel determines the port number to use based on the protocol.

Protocol	Default Port Number
SMTP	25
SMTPS	465
POP3	110
POP3S	995
IMAP	143
IMAPS	993

3.31.2. Options

Property	Default	Description
host		The host name or IP address to connect to.
port	See DefaultPorts	The TCP port number to connect on.
username		The user name on the email server.
password	null	The password on the email server.
ignoreUriScheme	false	If `false`, Camel uses the scheme to determine the transport protocol (POP, IMAP, SMTP etc.)
defaultEncoding	null	The default encoding to use for Mime Messages.
contentType	text/plain	The mail message content type. Use `text/html` for HTML mails.
folderName	INBOX	The folder to poll.
destination	username@host	**@deprecated** Use the `to` option instead. The `TO` recipients (receivers of the email).
to	username@host	The TO recipients (the receivers of the mail). Separate multiple email addresses with a comma.
replyTo	alias@host	The Reply-To recipients (the receivers of the response mail). Separate multiple email addresses with a comma.
CC	null	The CC recipients (the receivers of the mail). Separate multiple email addresses with a comma.
BCC	null	The BCC recipients (the receivers of the mail). Separate multiple email addresses with a comma.
from	camel@localhost	The FROM email address.
subject		The Subject of the message being sent. Note: Setting the subject in the header takes precedence over this option.
delete	false	Deletes the messages after they have been processed. This is done by setting the `DELETED` flag on the mail message. If `false`, the `SEEN` flag is set instead.
unseen	true	Is used to fetch only unseen messages (that is, new messages). Note that POP3 does not support the `SEEN` flag; use IMAP instead.

Property	Default	Description
`fetchSize`	`-1`	This option sets the maximum number of messages to consume during a poll. This can be used to avoid overloading a mail server, if a mailbox folder contains a lot of messages. Default value of `-1` means no fetch size and all messages will be consumed. Setting the value to 0 is a special corner case, where Camel will not consume any messages at all.
`alternativeBody-Header`	`CamelMailAlternat-iveBody`	Specifies the key to an IN message header that contains an alternative email body. For example, if you send emails in `text/html` format and want to provide an alternative mail body for non-HTML email clients, set the alternative mail body with this key as a header.
`debugMode`	`false`	It is possible to enable debug mode on the underlying mail framework. The SUN Mail framework logs the debug messages to `System.out` by default.
`connectionTimeout`	`30000`	The connection timeout can be configured in milliseconds. Default is 30 seconds.
`consumer.initialDelay`	`1000`	Milliseconds before the polling starts.
`consumer.delay`	`60000`	The default consumer delay is now 60 seconds. Camel will therefore only poll the mailbox once a minute to avoid overloading the mail server.
`consumer.useFixedDelay`	`false`	Set to `true` to use a fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
`disconnect`	`false`	Whether the consumer should disconnect after polling. If enabled this forces Camel to connect on each poll.
`mail.XXX`	`null`	You can set any additional java mail properties. For instance if you want to set a special property when using POP3 you can now provide the option directly in the URI such as: `mail.pop3.forgettopheaders=true`. You can set multiple such options, for example: `mail.pop3.forgettopheaders=true&` `mail.mime.encodefilename=true`.
`mapMailMessage`	`true`	Specifies whether Camel should map the received mail message to Camel body/headers. If set to true, the body of the mail message is mapped to the body of the Camel IN message and the mail headers are mapped to IN headers. If this option is set to false then the IN message contains a raw `javax.mail.Message`. You can retrieve this raw message by calling `exchange.getIn().getBody(javax.mail.Message.class)`.
`maxMessagesPerPoll`	`0`	Specifies the maximum number of messages to gather per poll. By default, no maximum is set. Can be used to set a limit of, for example, 1000 to avoid downloading thousands of files when the server starts up. Set a value of 0 or negative to disable this option.
`javaMailSender`	`null`	Specifies a pluggable `org.springframework.mail.javamail.JavaMailSender` instance in order to use a custom email implementation. If

Property	Default	Description
		none provided, Camel uses the default, `org.springframework.mail.javamail.` `JavaMailSenderImpl`.
`ignoreUnsupported-Charset`	false	Option to let Camel ignore unsupported charset in the local JVM when sending mails. If the charset is unsupported then `charset=XXX` (where `XXX` represents the unsupported charset) is removed from the `content-type` and it relies on the platform default instead.
`sslContextParameters`	null	Reference to a `org.apache.camel.util.jsse.SSLContextParameters` in the Registry. This reference overrides any configured SSLContextParameters at the component level. See Using the JSSE Configuration Utility for more information.

3.31.3. SSL support

The underlying mail framework is responsible for providing SSL support. Camel uses SUN JavaMail, which only trusts certificates issued by well known Certificate Authorities. So if you issue your own certificate, you have to import it into the local Java keystore file (see `SSLNOTES.txt` in JavaMail for details).

3.31.4. Mail Message Content

Camel uses the message exchange's IN body as the MimeMessage text content. The body is converted to `String.class`.

Camel copies all of the exchange's IN headers to the MimeMessage headers.

The subject of the MimeMessage can be configured using a header property on the IN message. The code below demonstrates this:

```
from("direct:a").setHeader("subject", constant(subject))
      .to("smtp://joe2@localhost");
```

The same applies for other MimeMessage headers such as recipients, so you can use a header property as `To` :

```
Map<String, Object> map = new HashMap<String, Object>();
map.put("To", "jenshansen@gmail.com");
map.put("From", "jbloggs@gmail.com");
map.put("Subject", "Camel rocks");

String body = "Hello Jens.\nYes it does.\n\nRegards Joe.";
template.sendBodyAndHeaders("smtp://jenshansen@gmail.com", body, map);
```

3.31.5. Headers take precedence over pre-configured recipients

The recipients specified in the message headers always take precedence over recipients pre-configured in the endpoint URI. The idea is that if you provide any recipients in the message headers, that is what you get. The recipients pre-configured in the endpoint URI are treated as a fallback.

In the sample code below, the email message is sent to jenshansen@gmail.com, because it takes precedence over the pre-configured recipient, info@mycompany.com. Any CC and BCC settings in the endpoint URI are also ignored and those recipients will not receive any mail. The choice between headers and pre-configured settings is all or nothing: the mail component *either* takes the recipients exclusively from the headers or exclusively from the pre-configured settings. It is not possible to mix and match headers and pre-configured settings.

```
Map<String, Object> headers = new HashMap<String, Object>();
        headers.put("to", "jenshansen@gmail.com");

        template.sendBodyAndHeaders(
                "smtp://admin@localhost?to=info@mycompany.com",
                "Hello World", headers);
```

3.31.6. Multiple recipients for easier configuration

It is possible to set multiple recipients using a comma-separated or a semicolon-separated list. This applies both to header settings and to settings in an endpoint URI. For example:

```
Map<String, Object> headers = new HashMap<String, Object>();
        headers.put("to",
            "jenshansen@gmail.com ; jbloggs@gmail.com ; janedoe@gmail.com");
```

The preceding example uses a semicolon, ;, as the separator character.

3.31.7. Setting sender name and email

You can specify recipients in the format, name <email>, to include both the name and the email address of the recipient.

For example, you define the following headers on the a Section 2.22, "Message" :

```
Map headers = new HashMap();
map.put("To", "Jens Hansen <jenshansen@gmail.com>");
map.put("From", "Joe Bloggs <jbloggs@gmail.com>");
map.put("Subject", "Camel is cool");
```

3.31.8. SUN JavaMail

SUN JavaMail is used under the hood for consuming and producing mails. We encourage end-users to consult these references when using either POP3 or IMAP protocol. Note particularly that POP3 has a much more limited set of features than IMAP, so end users are recommended to use IMAP where possible.

• SUN POP3 API

- SUN IMAP API

- And generally about the MAIL Flags

3.31.9. Samples

We start with a simple route that sends the messages received from a JMS queue as emails. The email account is the `admin` account on `mymailserver.com`.

```
from("jms://queue:subscription")
        .to("smtp://admin@mymailserver.com?password=secret");
```

In the next sample, we poll a mailbox for new emails once every minute. Notice that we use the special `consumer` option for setting the poll interval, `consumer.delay`, as 60000 milliseconds = 60 seconds.

```
from("imap://admin@mymailserver.com
     password=secret&unseen=true&consumer.delay=60000")
    .to("seda://mails");
```

In this sample we want to send a mail to multiple recipients.

```
// all the recipients of this mail are:
// To: camel@riders.org , easy@riders.org
// CC: me@you.org
// BCC: someone@somewhere.org
String recipients =
    "&To=camel@riders.org,easy@riders.org&
CC=me@you.org&BCC=someone@somewhere.org";

from("direct:a")
    .to("smtp://you@mymailserver.com?password=secret&From=you@apache.org"
        + recipients);
```

Check the Apache Camel website for several more examples, including handling mail attachments and SSL configuration.

3.32. Mock

Testing of distributed and asynchronous processing is notoriously difficult. The Section 3.32, "Mock", Section 3.49, "Test" and DataSet endpoints work great with the Camel Testing Framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's large range of Components together with the powerful Bean Integration.

The Mock component provides a powerful declarative testing mechanism, which is similar to jMock in that it allows declarative expectations to be created on any Mock endpoint before a test begins. Then the test is run, which typically fires messages to one or more endpoints, and finally the expectations can be asserted in a test case to ensure the system worked as expected.

This allows you to test various things like:

- The correct number of messages are received on each endpoint,

- The correct payloads are received, in the right order,

- Messages arrive on an endpoint in order, using some Expression to create an order testing function,

- Messages arrive match some kind of Predicate such as that specific headers have certain values, or that parts of the messages match some predicate, such as by evaluating an XPath or XQuery Expression.

Note that there is also the Test endpoint which is a Mock endpoint, but which uses a second endpoint to provide the list of expected message bodies and automatically sets up the Mock endpoint assertions. In other words, it is a Mock endpoint that automatically sets up its assertions from some sample messages in a Section 3.14, "File" or database, for example.

> ## ✖ Mock endpoints keep received Exchanges in memory indefinitely
>
> Remember that Mock is designed for testing. When you add Mock endpoints to a route, each Exchange sent to the endpoint will be stored (to allow for later validation) in memory until explicitly reset or the JVM is restarted. If you are sending high volume and/or large messages, this may cause excessive memory use. If your goal is to test deployable routes inline, consider using NotifyBuilder or AdviceWith in your tests instead of adding Mock endpoints to routes directly.
>
> From Camel 2.10 onwards there are two new options `retainFirst` and `retainLast` that can be used to limit the number of messages the Mock endpoints keep in memory.

3.32.1. URI format

```
mock:someName[?options]
```

where **someName** can be any string that uniquely identifies the endpoint.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.32.2. Options

Option	Default	Description
reportGroup	null	A size to use a throughput logger for reporting

3.32.3. Simple Example

Here's a simple example of Mock endpoint in use. First, the endpoint is resolved on the context. Then we set an expectation, and then, after the test has run, we assert that our expectations have been met.

```
MockEndpoint resultEndpoint =
      context.resolveEndpoint("mock:foo", MockEndpoint.class);

resultEndpoint.expectedMessageCount(2);

// send some messages
...

// now let's assert that the mock:foo endpoint received two messages
resultEndpoint.assertIsSatisfied();
```

You typically always call the assertIsSatisfied() method to test that the expectations were met after running a test.

Camel will by default wait 10 seconds when the `assertIsSatisfied()` is invoked. This can be configured by setting the `setResultWaitTime(milliseconds)` method.

When the assertion is satisfied then Camel will stop waiting and continue from the `assertIsSatisfied` method. That means if a new message arrives on the mock endpoint, just a bit later, that arrival will not affect the outcome of the assertion. Suppose you do want to test that no new messages arrives after a period thereafter, then you can do that by setting the `setAssertPeriod` method.

3.32.3.1. Using assertPeriod

When the assertion is satisfied then Camel will stop waiting and continue from the `assertIsSatisfied` method. That means if a new message arrives on the mock endpoint, just a bit later, that arrival will not affect the outcome of the assertion. Suppose you do want to test that no new messages arrives after a period thereafter, then you can do that by setting the `setAssertPeriod` method, for example:

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo",
    MockEndpoint.class);
resultEndpoint.setAssertPeriod(5000);
resultEndpoint.expectedMessageCount(2);

// send some messages
...

// now let's assert that the mock:foo endpoint received two messages
resultEndpoint.assertIsSatisfied();
```

3.32.4. Setting expectations

You can see from the javadoc of MockEndpoint the various helper methods you can use to set expectations. The main methods are as follows:

Method	Description
expectedMessageCount(int)	To define the expected message count on the endpoint.
expectedMinimumMessageCount(int)	To define the minimum number of expected messages on the endpoint.
expectedBodiesReceived(...)	To define the expected bodies that should be received (in order).
expectedHeaderReceived(...)	To define the expected header that should be received
expectsAscending(Expression)	To add an expectation that messages are received in order, using the given Expression to compare messages.
expectsDescending(Expression)	To add an expectation that messages are received in order, using the given Expression to compare messages.
expectsNoDuplicates(Expression)	To add an expectation that no duplicate messages are received; using an Expression to calculate a unique identifier for each message. This could be something like the `JMSMessageID` if using JMS, or some unique reference number within the message.

Here's another example:

```
resultEndpoint.expectedBodiesReceived("firstMessageBody", "secondMessageBody",
```

```
                                                  "thirdMessageBody");
```

3.32.4.1. Adding expectations to specific messages

In addition, you can use the message(int messageIndex) method to add assertions about a specific message that is received.

For example, to add expectations of the headers or body of the first message (using zero-based indexing like java.util.List), you can use the following code:

```
resultEndpoint.message(0).header("foo").isEqualTo("bar");
```

There are some examples of the Mock endpoint in use in the camel-core processor tests .

3.32.5. Mocking existing endpoints

Available as of Camel 2.7

Camel now allows you to automatically mock existing endpoints in your Camel routes.

How it works

Important: The endpoints are still in action, what happens is that a Section 3.32, "Mock" endpoint is injected and receives the message first, and then it delegates the message to the target endpoint. You can view this as a kind of intercept and delegate or endpoint listener.

Suppose you have the given route below:

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:start").to("direct:foo").to("log:foo").to(
                "mock:result");

            from("direct:foo").transform(constant("Bye World"));
        }
    };
}
```

You can then use the adviceWith feature in Camel to mock all the endpoints in a given route from your unit test, as shown below:

```
public void testAdvisedMockEndpoints() throws Exception {
    // advice the first route using the inlined AdviceWith route builder
    // which has extended capabilities than the regular route builder
    context.getRouteDefinitions().get(0)
          .adviceWith(context, new AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            // mock all endpoints
            mockEndpoints();
        }
```

```
    });

    getMockEndpoint("mock:direct:start").
        expectedBodiesReceived("Hello World");
    getMockEndpoint("mock:direct:foo").
        expectedBodiesReceived("Hello World");
    getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye World");
    getMockEndpoint("mock:result").expectedBodiesReceived("Bye World");

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();

    // additional test to ensure correct endpoints in registry
    assertNotNull(context.hasEndpoint("direct:start"));
    assertNotNull(context.hasEndpoint("direct:foo"));
    assertNotNull(context.hasEndpoint("log:foo"));
    assertNotNull(context.hasEndpoint("mock:result"));
    // all the endpoints was mocked
    assertNotNull(context.hasEndpoint("mock:direct:start"));
    assertNotNull(context.hasEndpoint("mock:direct:foo"));
    assertNotNull(context.hasEndpoint("mock:log:foo"));
}
```

Notice that the mock endpoints is given the uri `mock:<endpoint>`, for example `mock:direct:foo`. Camel logs at `INFO` level the endpoints being mocked:

```
INFO  Adviced endpoint [direct://foo] with mock endpoint [mock:direct:foo]
```

⚠ Mocked endpoints are without parameters

> Endpoints which are mocked will have their parameters stripped off. For example the endpoint "log:foo?showAll=true" will be mocked to the following endpoint "mock:log:foo". Notice the parameters have been removed.

It is also possible to only mock certain endpoints using a pattern. For example to mock all `log` endpoints you do as shown:

```
public void testAdvisedMockEndpointsWithPattern() throws Exception {
    // advice the first route using the inlined AdviceWith route builder
    // which has extended capabilities than the regular route builder
    context.getRouteDefinitions().get(0)
            .adviceWith(context, new AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            // mock only log endpoints
            mockEndpoints("log*");
        }
    });

    // now we can refer to log:foo as a mock and set our expectations
    getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye World");

    getMockEndpoint("mock:result").expectedBodiesReceived("Bye World");

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();
```

```
    // additional test to ensure correct endpoints in registry
    assertNotNull(context.hasEndpoint("direct:start"));
    assertNotNull(context.hasEndpoint("direct:foo"));
    assertNotNull(context.hasEndpoint("log:foo"));
    assertNotNull(context.hasEndpoint("mock:result"));
    // only the log:foo endpoint was mocked
    assertNotNull(context.hasEndpoint("mock:log:foo"));
    assertNull(context.hasEndpoint("mock:direct:start"));
    assertNull(context.hasEndpoint("mock:direct:foo"));
}
```

The pattern supported can be a wildcard or a regular expression. See more details about this functionality on the Apache Camel website.

> ⚠️ **Important**
>
> Mind that mocking endpoints causes the messages to be copied when they arrive on the mock. That means Camel will use more memory. This may not be suitable when you send in a lot of messages.

3.32.6. Limiting the number of messages to keep

The Mock endpoints will by default keep a copy of every Exchange that it received. So if you test with a lot of messages, then it will consume memory. From Camel 2.10 onwards we have introduced two options retainFirst and retainLast that can be used to specify to only keep N'th of the first and/or last Exchanges. For example in the code below, we only want to retain a copy of the first 5 and last 5 Exchanges the mock receives.

```
MockEndpoint mock = getMockEndpoint("mock:data");
mock.setRetainFirst(5);
mock.setRetainLast(5);
mock.expectedMessageCount(2000);
...
mock.assertIsSatisfied();
```

Using this has some limitations. The getExchanges() and getReceivedExchanges() methods on the MockEndpoint will return only the retained copies of the Exchanges. So in the example above, the list will contain 10 Exchanges; the first five, and the last five. The retainFirst and retainLast options also have limitations on which expectation methods you can use. For example the expectedXXX methods that work on message bodies, headers, etc. will operate only on the retained messages. In the example above they can test only the expectations on the 10 retained messages.

3.32.7. Testing with arrival times

The Section 3.32, "Mock" endpoint stores the arrival time of the message as a property on the Exchange.

```
Date time = exchange.getProperty(Exchange.RECEIVED_TIMESTAMP, Date.class);
```

You can use this information to know when the message arrived on the mock. But it also provides foundation to know the time interval between the previous and next message arrived on the mock. You can use this to set expectations using the arrives DSL on the Section 3.32, "Mock" endpoint.

For example to say that the first message should arrive between 0-2 seconds before the next you can do:

```
mock.message(0).arrives().noLaterThan(2).seconds().beforeNext();
```

You can also define this as that the second message (0 index based) should arrive no later than 0-2 seconds after the previous:

```
mock.message(1).arrives().noLaterThan(2).seconds().afterPrevious();
```

You can also use between to set a lower bound. For example suppose that it should be between 1-4 seconds:

```
mock.message(1).arrives().between(1, 4).seconds().afterPrevious();
```

You can also set the expectation on all messages, for example to say that the gap between them should be at most 1 second:

```
mock.allMessages().arrives().noLaterThan(1).seconds().beforeNext();
```

> **time units**
>
> In the example above we use `seconds` as the time unit, but Camel offers `milliseconds`, and `minutes` as well.

3.33. MyBatis

The **MyBatis** component allows you to query, poll, insert, update and delete data in a relational database using MyBatis.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-mybatis</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.33.1. URI format

```
mybatis:statementName[?options]
```

Where **statementName** is the statement name in the MyBatis XML configuration file which maps to the query, insert, update or delete operation you wish to evaluate.

You can append query options to the URI in the following format, `?option=value&option=value&...`

This component will by default load the MyBatis SqlMapConfig file from the root of the classpath with the expected name of `SqlMapConfig.xml`. If the file is located in another location, you will need to configure the configurationUri option on the MyBatisComponent component.

3.33.2. Options

Option	Type	Default	Description
consumer.onConsume	String	null	Statements to run after consuming. Can be used, for example, to update rows after

Option	Type	Default	Description
			they have been consumed and processed in Camel. Multiple statements can be separated with commas.
`consumer. useIterator`	`boolean`	`true`	If `true` each row returned when polling will be processed individually. If `false` the entire `List` of data is set as the IN body.
`consumer. routeEmptyResultSet`	`boolean`	`false`	Sets whether empty result sets should be routed.
`statementType`	`StatementType`	`null`	Mandatory to specify for the Producer to control which kind of operation to invoke. The enum values are: `SelectOne`, `SelectList`, `Insert`, `InsertList`, `Update`, `Delete`.
`maxMessagesPerPoll`	`int`	`0`	An integer to define the maximum messages to gather per poll. By default, no maximum is set. Can be used to set a limit of, for example, 1000 to avoid when starting up the server that there are thousands of files. Set a value of 0 or negative to disable it.

3.33.3. Message Headers

Camel will populate the result message, either IN or OUT with a header with the statement used:

Header	Type	Description
`CamelMyBatis-StatementName`	`String`	The **statementName** used (for example: insertAccount).
`CamelMyBatisResult`	`Object`	The **response** returned from MyBatis in any of the operations. For instance an `INSERT` could return the auto-generated key, or number of rows etc.

3.33.4. Message Body

The response from MyBatis will only be set as body if it is a `SELECT` statement. That means, for example, for `INSERT` statements Camel will not replace the body. This allows you to continue routing and keep the original body. The response from MyBatis is always stored in the header with the key `CamelMyBatisResult`.

3.33.5. Samples

For example if you wish to consume beans from a JMS queue and insert them into a database you could do the following:

```
from("activemq:queue:newAccount")
    .to("mybatis:insertAccount?statementType=Insert");
```

Notice we have to specify the `statementType`, as we need to instruct Camel which kind of operation to invoke. The **insertAccount** value given above is the MyBatis ID in the SQL map file:

```
<!-- Insert example, using the Account parameter class -->
   <insert id="insertAccount" parameterClass="Account">
      insert into ACCOUNT (
         ACC_ID,
         ACC_FIRST_NAME,
         ACC_LAST_NAME,
         ACC_EMAIL
      )
      values (
         #id#, #firstName#, #lastName#, #emailAddress#
      )
   </insert>
```

3.33.6. Using StatementType for better control of MyBatis

When routing to an MyBatis endpoint you will want more fine grained control so you can control whether the SQL statement to be executed is a `SELECT`, `UPDATE`, `DELETE` or `INSERT` etc. So for instance if we want to route to an MyBatis endpoint in which the IN body contains parameters to a `SELECT` statement we can do:

```
from("direct:start")
    .to("mybatis:selectAccountById?statementType=QueryForObject")
    .to("mock:result");
```

In the code above we invoke the MyBatis statement `selectAccountById` and the IN body should contain the account id we want to retrieve, such as an `Integer` type.

We can do the same for some of the other operations, such as `SelectList`:

```
from("direct:start")
    .to("mybatis:selectAllAccounts?statementType=SelectList")
    .to("mock:result");
```

And the same for `UPDATE`, where we can send an `Account` object as the IN body to MyBatis:

```
from("direct:start")
    .to("mybatis:updateAccount?statementType=Update")
    .to("mock:result");
```

3.33.6.1. Using onConsume

This component supports executing statements **after** data have been consumed and processed by Camel. This allows you to do post updates in the database. Notice all statements must be `UPDATE` statements. Camel supports executing multiple statements whose names should be separated by commas.

The route below illustrates executing the **consumeAccount** statement after the data is processed. This allows us to change the status of the row in the database to "processed", so we avoid consuming it twice or more.

```
from("mybatis:selectUnprocessedAccounts?consumer.
```

```
onConsume=consumeAccount").to("mock:results");
```

And the statements in the sqlmap file:

```
<select id="selectUnprocessedAccounts" resultMap="AccountResult">
   select * from ACCOUNT where PROCESSED = false
</select>
<update id="consumeAccount" parameterClass="Account">
   update ACCOUNT set PROCESSED = true where ACC_ID = #id#
</update>
```

3.34. Properties

3.34.1. Properties Component

3.34.1.1. URI format

```
properties:key[?options]
```

where **key** is the key for the property to lookup

3.34.1.2. Options

Name	Type	Default	Description
cache	boolean	true	Whether or not to cache loaded properties.
locations	String	null	A list of locations to load properties. You can use comma to separate multiple locations. This option will override any default locations and **only** use the locations from this option.

3.34.2. Using PropertyPlaceholder

Camel now provides a new `PropertiesComponent` in **camel-core** which allows you to use property placeholders when defining Camel Endpoint URIs. This works much like you would do if using Spring's `<property-placeholder>` tag. However Spring have a limitation which prevents 3rd party frameworks to leverage Spring property placeholders to the fullest. See more at How do I use Spring Property Placeholder with Camel XML .

The property placeholder is generally in use when doing:

- lookup or creating endpoints

- lookup of beans in the Registry

- additional supported in Spring XML (see below in examples)

- using Blueprint PropertyPlaceholder with Camel Section 3.34, "Properties" component

3.34.2.1. Syntax

The syntax to use Camel's property placeholder is to use {{ key }} for example {{ file.uri }} where file.uri is the property key. You can use property placeholders in parts of the endpoint URI's which for example you can use placeholders for parameters in the URIs.

3.34.2.2. PropertyResolver

As usual Camel provides a pluggable mechanism which allows 3rd part to provide their own resolver to lookup properties. Camel provides a default implementation org.apache.camel.component.properties.DefaultPropertiesResolver which is capable of loading properties from the file system, classpath or Registry. You can prefix the locations with either:

- ref: to lookup in the Registry

- file: to load the from file system

- classpath: to load from classpath (this is also the default if no prefix is provided)

- blueprint: **Camel 2.7:** to use a specific OSGi blueprint placeholder service

3.34.2.3. Defining location

The PropertiesResolver need to know a location(s) where to resolve the properties. You can define one to many locations. If you define the location in a single String property you can separate multiple locations with comma such as:

```
pc.setLocation(
    "com/mycompany/myprop.properties,com/mycompany/other.properties");
```

Using system and environment variables in locations

Available as of Camel 2.7

The location now supports using placeholders for JVM system properties and OS environments variables.

For example:

```
location=file:${karaf.home}/etc/foo.properties
```

In the location above we defined a location using the file scheme using the JVM system property with key karaf.home.

To use an OS environment variable instead you would have to prefix with env:

```
location=file:${env:APP_HOME}/etc/foo.properties
```

where `APP_HOME` is an OS environment.

You can have multiple placeholders in the same location, such as:

```
location=file:${env:APP_HOME}/etc/${prop.name}.properties
```

3.34.2.4. Configuring in Java DSL

You have to create and register the `PropertiesComponent` under the name `properties` such as:

```
PropertiesComponent pc = new PropertiesComponent();
pc.setLocation("classpath:com/mycompany/myprop.properties");
context.addComponent("properties", pc);
```

3.34.2.5. Configuring in Spring XML

Spring XML offers two variations to configure. You can define a Spring bean as a `PropertiesComponent` which resembles the way done in Java DSL. Or you can use the `<propertyPlaceholder>` tag.

```
<bean id="properties"
      class="org.apache.camel.component.properties.PropertiesComponent">
   <property name="location"
      value="classpath:com/mycompany/myprop.properties"/>
</bean>
```

Using the `<propertyPlaceholder>` tag makes the configuration a bit more fresh such as:

```
<camelContext ...>
   <propertyPlaceholder id="properties"
         location="com/mycompany/myprop.properties"/>
</camelContext>
```

3.34.2.6. Using a Properties from the Registry

For example in OSGi you may want to expose a service which returns the properties as a `java.util.Properties` object.

Then you could setup the Section 3.34, "Properties" component as follows:

```
<propertyPlaceholder id="properties" location="ref:myProperties"/>
```

where `myProperties` is the id to use for lookup in the OSGi registry. Notice we use the `ref:` prefix to tell Camel that it should lookup the properties for the Registry.

3.34.2.7. Examples using properties component

When using property placeholders in the endpoint URIs you can either use the `properties:` component or define the placeholders directly in the URI. We will show example of both cases, starting with the former.

```
// properties
cool.end=mock:result
```

```
// route
from("direct:start").to("properties:{{cool.end}}");
```

You can also use placeholders as a part of the endpoint uri:

```
// properties
cool.foo=result
```

```
// route
from("direct:start").to("properties:mock:{{cool.foo}}");
```

In the example above the to endpoint will be resolved to mock:result.

You can also have properties with refer to each other such as:

```
// properties
cool.foo=result
cool.concat=mock:{{cool.foo}}
```

```
// route
from("direct:start").to("properties:mock:{{cool.concat}}");
```

Notice how cool.concat refer to another property.

The properties: component also offers you to override and provide a location in the given uri using the locations option:

```
from("direct:start")
    .to("properties:bar.end?locations=com/mycompany/bar.properties");
```

3.34.2.8. Examples

You can also use property placeholders directly in the endpoint uris without having to use properties:.

```
// properties
cool.foo=result
```

```
// route
from("direct:start").to("mock:{{cool.foo}}");
```

And you can use them in multiple wherever you want them:

```
// properties
cool.start=direct:start
cool.showid=true
cool.result=result
```

```
// route
from("{{cool.start}}")
    .to("log:{{cool.start}}?showBodyType=false"
    + "&showExchangeId={{cool.showid}}")
    .to("mock:{{cool.result}}");
```

You can also your property placeholders when using ProducerTemplate for example:

```
template.sendBody("{{cool.start}}", "Hello World");
```

3.34.2.9. Example with Simple language

The Simple language now also support using property placeholders, for example in the route below:

```
// properties
cheese.quote=Camel rocks

// route
from("direct:start")
    .transform().simple(
    "Hi ${body} do you think ${properties:cheese.quote}?");
```

You can also specify the location in the Simple language for example:

```
// bar.properties
bar.quote=Beer tastes good

// route
from("direct:start")
   .transform()
   .simple(
      "Hi ${body}. ${properties:com/mycompany/bar.properties:bar.quote}.");
```

3.34.2.10. Additional property placeholder supported in Spring XML

The property placeholders is also supported in many of the Camel Spring XML tags such as `<package>`, `<packageScan>`, `<contextScan>`, `<jmxAgent>`, `<endpoint>`, `<routeBuilder>`, `<proxy>` and the others.

The example below has property placeholder in the <jmxAgent> tag:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
   <propertyPlaceholder id="properties"
      location="org/apache/camel/spring/jmx.properties"/>

   <!-- we can use propery placeholders when we define the JMX agent -->
   <jmxAgent id="agent"
        registryPort="{{myjmx.port}}" disabled="{{myjmx.disabled}}"
      usePlatformMBeanServer="{{myjmx.usePlatform}}"
      createConnector="true"
      statisticsLevel="RoutesOnly"/>

   <route id="foo" autoStartup="false">
      <from uri="seda:start"/>
      <to uri="mock:result"/>
   </route>
</camelContext>
```

You can also define property placeholders in the various attributes on the <camelContext> tag such as trace as shown here:

```
<camelContext trace="{{foo.trace}}"
        xmlns="http://camel.apache.org/schema/spring">
    <propertyPlaceholder
        id="properties"
        location="org/apache/camel/spring/processor/myprop.properties"/>

    <template id="camelTemplate" defaultEndpoint="{{foo.cool}}"/>

    <route>
        <from uri="direct:start"/>
        <setHeader headerName="{{foo.header}}">
            <simple>${in.body} World!</simple>
        </setHeader>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

3.34.2.11. Overriding a property setting using a JVM System Property

It is possible to override a property value at runtime using a JVM System property without the need to restart the application to pick up the change. This may also be accomplished from the command line by creating a JVM System property of the same name as the property it replaces with a new value. An example of this is given below

```
PropertiesComponent pc =
      context.getComponent("properties", PropertiesComponent.class);
pc.setCache(false);

System.setProperty("cool.end", "mock:override");
System.setProperty("cool.result", "override");

context.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").to("properties:cool.end");
        from("direct:foo").to("properties:mock:{{cool.result}}");
    }
});
context.start();

getMockEndpoint("mock:override").expectedMessageCount(2);

template.sendBody("direct:start", "Hello World");
template.sendBody("direct:foo", "Hello Foo");

System.clearProperty("cool.end");
System.clearProperty("cool.result");
```

```
assertMockEndpointsSatisfied();
```

3.34.2.12. Using property placeholders for any kind of attribute in the XML DSL

Available as of Camel 2.7

Previously it was only the `xs:string` type attributes in the XML DSL that support placeholders. For example often a timeout attribute would be a `xs:int` type and thus you cannot set a string value as the placeholder key. This is now possible from Camel 2.7 onwards using a special placeholder namespace.

In the example below we use the `prop` prefix for the namespace `http://camel.apache.org/schema/placeholder` by which we can use the `prop` prefix in the attributes in the XML DSLs. Notice how we use that in the Section 2.32, "Multicast" to indicate that the option `stopOnException` should be the value of the placeholder with the key "stop".

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:prop="http://camel.apache.org/schema/placeholder"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
             http://www.springframework.org/schema/beans/spring-beans.xsd
http://camel.apache.org/schema/spring
             http://camel.apache.org/schema/spring/camel-spring.xsd
    ">

    <!-- Notice in the declaration above, we have defined the prop -->
    <!-- prefix as the Camel placeholder namespace -->

    <bean id="damn" class="java.lang.IllegalArgumentException">
       <constructor-arg index="0" value="Damn"/>
    </bean>

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

    <propertyPlaceholder id="properties" location=
       "classpath:org/apache/camel/component/properties/myprop.properties"
       xmlns="http://camel.apache.org/schema/spring"/>

       <route>
            <from uri="direct:start"/>
            <!-- use prop namespace, to define a property placeholder,
                 which maps to option stopOnException={{stop}} -->
            <multicast prop:stopOnException="stop">
                <to uri="mock:a"/>
                <throwException ref="damn"/>
                <to uri="mock:b"/>
            </multicast>
        </route>

    </camelContext>

</beans>
```

In our properties file we have the value defined as

```
stop=true
```

3.34.2.13. Using property placeholder in the Java DSL

Available as of Camel 2.7

Likewise we have added support for defining placeholders in the Java DSL using the new `placeholder` DSL as shown in the following equivalent example:

```
from("direct:start")
    // use a property placeholder for the option stopOnException on the
    // Multicast EIP which should have the value of {{stop}}
    // key being looked up in the properties file
    .multicast()
        .placeholder("stopOnException", "stop")
        .to("mock:a")
        .throwException(new IllegalAccessException("Damn"))
        .to("mock:b");
```

3.34.2.14. Using Blueprint property placeholder with Camel routes

Available as of Camel 2.7

Camel supports Blueprint which also offers a property placeholder service. Camel supports convention over configuration, so all you have to do is to define the OSGi Blueprint property placeholder in the XML file as shown below:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
   xsi:schemaLocation="
http://www.osgi.org/xmlns/blueprint/v1.0.0
           http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

   <!-- OSGI blueprint property placeholder -->
   <cm:property-placeholder id="myblueprint.placeholder"
        persistent-id="camel.blueprint">
     <!-- list some properties for this test -->
     <cm:default-properties>
        <cm:property name="result" value="mock:result"/>
     </cm:default-properties>
   </cm:property-placeholder>

   <camelContext xmlns="http://camel.apache.org/schema/blueprint">

   <!-- in the route we can use {{ }} placeholders which will lookup in -->
   <!-- blueprint as Camel will auto detect the OSGi blueprint property -->
   <!-- placeholder and use it -->
     <route>
        <from uri="direct:start"/>
        <to uri="mock:foo"/>
```

```
        <to uri="{{result}}"/>
      </route>
   </camelContext>
</blueprint>
```

By default Camel detects and uses OSGi blueprint property placeholder service. You can disable this by setting the attribute `useBlueprintPropertyResolver` to false on the `<camelContext>` definition.

About placeholder syntaxes

Notice how we can use the Camel syntax for placeholders {{ }} in the Camel route, which will lookup the value from OSGi blueprint. The blueprint syntax for placeholders is ${ }. So outside the <camelContext> you must use the ${ } syntax. Whereas inside <camelContext> you must use {{ }} syntax. OSGi blueprint allows you to configure the syntax, so you can align those if you want.

You can also explicit refer to a specific OSGi blueprint property placeholder by its id. For that you need to use the Camel's <propertyPlaceholder> as shown in the example below:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
   xsi:schemaLocation="
http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/
blueprint/v1.0.0/blueprint.xsd">

   <!-- OSGI blueprint property placeholder -->
   <cm:property-placeholder id="myblueprint.placeholder"
         persistent-id="camel.blueprint">
      <!-- list some properties for this test -->
      <cm:default-properties>
         <cm:property name="result" value="mock:result"/>
      </cm:default-properties>
   </cm:property-placeholder>

   <camelContext xmlns="http://camel.apache.org/schema/blueprint">

      <!-- using Camel properties component and refer to the blueprint  -->
      <!-- property placeholder by its id -->
      <propertyPlaceholder id="properties"
            location="blueprint:myblueprint.placeholder"/>

      <!-- in the route we can use {{ }} placeholders which will lookup -->
      <!-- in blueprint -->
      <route>
         <from uri="direct:start"/>
         <to uri="mock:foo"/>
         <to uri="{{result}}"/>
      </route>

   </camelContext>

</blueprint>
```

Notice how we use the `blueprint` scheme to refer to the OSGi blueprint placeholder by its id. This allows you to mix and match, for example you can also have additional schemes in the location. For example to load a file from the classpath you can do:

```
location="blueprint:myblueprint.placeholder,
   classpath:myproperties.properties"
```

Each location is separated by comma.

3.35. Quartz

The **quartz:** component provides a scheduled delivery of messages using the Quartz scheduler. Each endpoint represents a different timer (in Quartz terms, a Trigger and JobDetail).

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-quartz</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

⚠️ **Using cron expressions**

> Configuring the cron expression is based on a URI option. Note: it is possible to use the / cron special character (for increments). However, you may need to escape certain URI characters such as using ? in the quartz cron expression.

3.35.1. URI format

```
quartz://timerName?options
quartz://groupName/timerName?options
quartz://groupName/timerName/cronExpression       (@deprecated)
quartz://groupName/timerName/?cron=expression    (Camel 2.0)
quartz://timerName?cron=expression               (Camel 2.0)
```

The component uses either a `CronTrigger` or a `SimpleTrigger`. If no cron expression is provided, the component uses a simple trigger. If no `groupName` is provided, the quartz component uses the `Camel` group name.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.35.2. Options

Parameter	Default	Description
cron	*None*	Specifies a `cron` expression (not compatible with the `trigger.*` or `job.*` options).
trigger.repeatCount	0	SimpleTrigger: How many times should the timer repeat?
trigger.repeatInterval	0	SimpleTrigger: The amount of time in milliseconds between repeated triggers.

Parameter	Default	Description
job.name	null	Sets the job name.
job.XXX	null	Sets the job option with the XXX setter name.
trigger.XXX	null	Sets the trigger option with the XXX setter name.
stateful	false	Uses a Quartz StatefulJob instead of the default job.
fireNow	false	If it is true will fire the trigger when the route is start when using SimpleTrigger.

For example, the following routing rule will fire two timer events to the `mock:results` endpoint:

```
from(
    "quartz://myGroup/myTimerName?trigger.repeatInterval=2"
        + "&trigger.repeatCount=1")
        .routeId("myRoute").to("mock:result");
```

When using a StatefulJob, the JobDataMap is re-persisted after every execution of the job, thus preserving state for the next execution.

3.35.3. Configuring quartz.properties file

By default Quartz will look for a `quartz.properties` file in the root of the classpath. If you are using WAR deployments this means just drop the quartz.properties in `WEB-INF/classes`.

However the Camel Section 3.35, "Quartz" component also allows you to configure properties:

Parameter	Default	Type	Description
properties	null	Properties	You can configure a `java.util.Propoperties` instance.
propertiesFile	null	String	File name of the properties to load from the classpath

To do this you can configure this in Spring XML as follows

```
<bean id="quartz"
    class="org.apache.camel.component.quartz.QuartzComponent">
    <property name="propertiesFile"
    value="com/mycompany/myquartz.properties"/>
</bean>
```

3.35.4. Starting the Quartz scheduler

The Section 3.35, "Quartz" component offers an option to let the Quartz scheduler be started delayed, or not auto started at all.

Parameter	Default	Type	Description
startDelayedSeconds	0	int	Seconds to wait before starting the quartz scheduler.

Parameter	Default	Type	Description
autoStartScheduler	true	boolean	Whether or not the scheduler should be auto started.

To do this you can configure this in Spring XML as follows

```
<bean id="quartz"
    class="org.apache.camel.component.quartz.QuartzComponent">
    <property name="startDelayedSeconds" value="5"/>
</bean>
```

3.35.5. Clustering

If you use Quartz in clustered mode, for example, the `JobStore` is clustered. Then from Camel 2.4 onwards the Section 3.35, "Quartz" component will **not** pause/remove triggers when a node is being stopped/shutdown. This allows the trigger to keep running on the other nodes in the cluster.

Note : When running in clustered node no checking is done to ensure unique job name/group for endpoints.

3.35.6. Message Headers

Camel adds the getters from the Quartz Execution Context as header values. The following headers are added: `calendar`, `fireTime`, `jobDetail`, `jobInstance`, `jobRuntTime`, `mergedJobDataMap`, `nextFireTime`, `previousFireTime`, `refireCount`, `result`, `scheduledFireTime`, `scheduler`, `trigger`, `triggerName`, `triggerGroup`.

The `fireTime` header contains the `java.util.Date` of when the exchange was fired.

3.35.7. Using Cron Triggers

Quartz supports Cron-like expressions for specifying timers in a handy format. You can use these expressions in the `cron` URI parameter; though to preserve valid URI encoding we allow + to be used instead of spaces. Quartz provides a little tutorial on how to use cron expressions.

For example, the following will fire a message every five minutes starting at 12pm (noon) to 6pm on weekdays:

```
from("quartz://myGroup/myTimerName?cron=0+0/5+12-18+?+*+MON-FRI")
    .to("activemq:Totally.Rocks");
```

which is equivalent to using the cron expression

```
0 0/5 12-18 ? * MON-FRI
```

The following table shows the URI character encodings we use to preserve valid URI syntax:

URI Character	Cron character
+	*Space*

3.36. Ref

The **ref:** component is used for lookup of existing endpoints bound in the Registry.

3.36.1. URI format

```
ref:someName
```

where **someName** is the name of an endpoint in the Registry (usually, but not always, the Spring registry). If you are using the Spring registry, `someName` would be the bean ID of an endpoint in the Spring registry.

3.36.2. Runtime lookup

This component can be used when you need dynamic discovery of endpoints in the Registry where you can compute the URI at runtime. Then you can look up the endpoint using the following code:

```
// lookup the endpoint
   String myEndpointRef = "bigspenderOrder";
   Endpoint endpoint = context.getEndpoint("ref:" + myEndpointRef);

   Producer producer = endpoint.createProducer();
   Exchange exchange = producer.createExchange();
   exchange.getIn().setBody(payloadToSend);
   // send the exchange
   producer.process(exchange);
   ...
```

And you could have a list of endpoints defined in the Registry such as:

```
<camelContext id="camel"
      xmlns="http://activemq.apache.org/camel/schema/spring">
   <endpoint id="normalOrder" uri="activemq:order.slow"/>
   <endpoint id="bigspenderOrder" uri="activemq:order.high"/>
      ...
</camelContext>
```

3.36.3. Sample

In the sample below we use the `ref:` in the URI to reference the endpoint with the Spring ID, `endpoint2`:

```
<bean id="mybean" class="org.apache.camel.spring.example.DummyBean">
  <property name="endpoint" ref="endpoint1"/>
</bean>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <jmxAgent id="agent" disabled="true"/>
  <endpoint id="endpoint1" uri="direct:start"/>
  <endpoint id="endpoint2" uri="mock:end"/>

  <route>
```

```
    <from ref="endpoint1"/>
    <to uri="ref:endpoint2"/>
  </route>
</camelContext>
```

You could, of course, have used the `ref` attribute instead:

```
<to ref="endpoint2"/>
```

Which is the more common way to write it.

3.37. RMI

The **rmi:** component binds PojoExchanges to the RMI protocol (JRMP).

Since this binding is just using RMI, normal RMI rules still apply regarding what methods can be invoked. This component supports only PojoExchanges that carry a method invocation from an interface that extends the Remote interface. All parameters in the method should be either Serializable or `Remote` objects.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-rmi</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.37.1. URI format

```
rmi://rmi-regisitry-host:rmi-registry-port/registry-path[?options]
```

For example:

```
rmi://localhost:1099/path/to/service
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.37.2. Options

Name	Default Value	Description
method	null	You can set the name of the method to invoke.
remoteInterfaces	null	It is now possible to use this option from **Camel 2.7:** in the XML DSL. It can be a list of interface names separated by comma.

3.37.3. Using

To call out to an existing RMI service registered in an RMI registry, create a route similar to the following:

```
from("pojo:foo").to("rmi://localhost:1099/foo");
```

To bind an existing Camel processor or service in an RMI registry, define an RMI endpoint as follows:

```
RmiEndpoint endpoint= (RmiEndpoint) endpoint("rmi://localhost:1099/bar");
endpoint.setRemoteInterfaces(ISay.class);
from(endpoint).to("pojo:bar");
```

Note that when binding an RMI consumer endpoint, you must specify the `Remote` interfaces exposed.

In XML DSL you can do as follows from **Camel 2.7** onwards:

```
<camel:route>
        <from uri="rmi://localhost:37541/helloServiceBean?remoteInterfaces=
org.apache.camel.example.osgi.HelloService"/>
        <to uri="bean:helloServiceBean"/>
    </camel:route>
```

3.38. RSS

The **rss:** component is used for polling RSS feeds. Camel will default poll the feed every 60th seconds.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-rss</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

The RSS component ships with an RSS dataformat that can be used to convert between String (as XML) and ROME RSS model objects, as well as filter out certain entries. Camel's Bean Integration can also be used for filtering out RSS entries. See the Camel Website for examples of this component in use.

Note: The component currently only supports polling (consuming) feeds.

3.38.1. URI format

```
rss:rssUri
```

where `rssUri` is the URI to the RSS feed to poll.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.38.2. Options

Property	Default	Description
splitEntries	true	If `true`, Camel splits a feed into its individual entries and returns each entry, poll by poll. For example, if a feed contains seven entries, Camel returns the first entry on the first poll, the second entry on the second poll, and

Property	Default	Description
		so on. When no more entries are left in the feed, Camel contacts the remote RSS URI to obtain a new feed. If `false`, Camel obtains a fresh feed on every poll and returns all of the feed's entries.
filter	true	Use in combination with the `splitEntries` option in order to filter returned entries. By default, Camel applies the `UpdateDateFilter` filter, which returns only new entries from the feed, ensuring that the consumer endpoint never receives an entry more than once. The filter orders the entries chronologically, with the newest returned last.
throttleEntries	true	Sets whether all entries identified in a single feed poll should be delivered immediately. If true, only one entry is processed per consumer.delay. Only applicable when splitEntries is set to true.
lastUpdate	null	Use in combination with the `filter` option to block entries earlier than a specific date/time (uses the `entry.updated` timestamp). The format is: `yyyy-MM-ddTHH:MM:ss`. Example: `2007-12-24T17:45:59`.
feedHeader	true	Specifies whether to add the ROME `SyndFeed` object as a header.
sortEntries	false	If `splitEntries` is `true`, this specifies whether to sort the entries by updated date.
consumer.delay	60000	Delay in milliseconds between each poll.
consumer.initialDelay	1000	Milliseconds before polling starts.
consumer.userFixedDelay	false	Set to `true` to use fixed delay between pools, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.

3.38.3. Exchange data types

Camel initializes the In body on the Exchange with a ROME `SyndFeed`. Depending on the value of the `splitEntries` flag, Camel returns either a `SyndFeed` with one `SyndEntry` or a `java.util.List` of `SyndEntrys`.

Option	Value	Behavior
splitEntries	true	A single entry from the current feed is set in the exchange.
splitEntries	false	The entire list of entries from the current feed is set in the exchange.

3.38.4. Message Headers

Header	Description
CamelRssFeed	The entire `SyncFeed` object.

3.39. SEDA

The **seda:** component provides asynchronous SEDA behavior, so that messages are exchanged on a BlockingQueue and consumers are invoked in a separate thread from the producer.

Note that queues are only visible within a *single* CamelContext. If you want to communicate across CamelContext instances (for example, communicating between Web applications), see the Section 3.52, "VM" component.

This component does not implement any kind of persistence or recovery, if the VM terminates while messages are yet to be processed. If you need persistence, reliability or distributed SEDA, try using either Section 3.25, "JMS" or Section 3.1, "ActiveMQ".

> ### Synchronous
>
> The Section 3.11, "Direct" component provides synchronous invocation of any consumers when a producer sends a message exchange.

3.39.1. URI format and options

```
seda:someName[?options]
```

where **someName** can be any string that uniquely identifies the endpoint within the current CamelContext.

You can append query options to the URI in the following format, `?option=value&option=value&...`

Note: the same queue name must be used for both producer and consumer.

An exactly identical Section 3.39, "SEDA" queue name **must** be used for both the producer endpoint and the consumer endpoint. Otherwise Camel will create a second Section 3.39, "SEDA" endpoint, even though the someName portion of the queue is identical. For example:

```
from("direct:foo").to("seda:bar?concurrentConsumers=5");
```

```
from("seda:bar?concurrentConsumers=5").to("file://output");
```

Options

Name	Default	Description
size		The maximum size (= capacity of the number of messages it can max hold) of the SEDA queue. The size is unbounded by default.
concurrent-Consumers	1	the number of concurrent threads to process exchanges.
waitForTaskTo-Complete	IfReplyExpected	option to specify whether the caller should wait for the async task to complete or not before continuing. The following three options are supported: Always, Never or IfReplyExpected. The first two values are self-explanatory. The last value, IfReplyExpected, will only wait if the message is Section 2.39, "Request Reply" based. The default option is IfReplyExpected. See more information about Async messaging.

Name	Default	Description
`timeout`	`30000`	Timeout in milliseconds a seda producer will at most waiting for an async task to complete. See `waitForTaskToComplete` and Async for more details. You can disable timeout by using 0 or a negative value.
`multipleConsumers`	`false`	Specifies whether multiple consumers is allowed or not. If enabled you can use Section 3.39, "SEDA" for a pubsub style messaging. Send a message to a seda queue and have multiple consumers receive a copy of the message. This option should be specified on every consumer endpoint, if in use.
`limitConcurrent-Consumers`	`true`	Whether to limit the concurrentConsumers to maximum 500. If it is configured with a higher number an exception will be thrown. You can disable this check by turning this option off.
`blockWhenFull`	false	Whether to block the current thread when sending a message to a SEDA endpoint, and the SEDA queue is full (capacity hit). By default an exception will be thrown stating the queue is full. By setting this option to `true` the caller thread will instead block and wait until the message can be delivered to the SEDA queue.
`queueSize`		The maximum size (capacity of the number of messages it can hold) of the SEDA queue.
`pollTimeout`	1000	Consumer only. The timeout used when polling. When a timeout occurs then the consumer can check whether its allowed to continue to run. Setting a lower value allows the consumer to react faster upon shutting down.

See the Camel Website for the most up-to-date examples of this component in use.

3.39.2. Use of Request Reply

The Section 3.39, "SEDA" component supports using Section 2.39, "Request Reply", where the caller will wait for the Async route to complete. For instance:

```
from("mina:tcp://0.0.0.0:9876?textline=true&sync=true").to("seda:input");

from("seda:input").to("bean:processInput").to("bean:createResponse");
```

In the route above, we have a TCP listener on port 9876 that accepts incoming requests. The request is routed to the `seda:input` queue. As it is a Section 2.39, "Request Reply" message, we wait for the response. When the consumer on the `seda:input` queue is complete, it copies the response to the original message response.

Using Section 2.39, "Request Reply" over Section 3.39, "SEDA" or Section 3.52, "VM", you can chain as many endpoints as you like.

3.39.3. Concurrent consumers

By default, the SEDA endpoint uses a single consumer thread, but you can configure it to use concurrent consumer threads. So instead of thread pools you can use:

```
from("seda:stageName?concurrentConsumers=5").process(...)
```

As for the difference between the two, note a thread pool can increase/shrink dynamically at runtime depending on load, whereas the number of concurrent consumers is always fixed.

3.39.4. Thread pools

Be aware that adding a thread pool to a SEDA endpoint by doing something like:

```
from("seda:stageName").thread(5).process(...)
```

Can wind up with two `BlockQueues` : one from the SEDA endpoint, and one from the workqueue of the thread pool, which may not be what you want. Instead, you might wish to configure a Section 3.11, "Direct" endpoint with a thread pool, which can process messages both synchronously and asynchronously. For example:

```
from("direct:stageName").thread(5).process(...)
```

You can also directly configure number of threads that process messages on a SEDA endpoint using the `concurrentConsumers` option.

3.40. Servlet

The **servlet:** component provides HTTP based endpoints for consuming HTTP requests that arrive at a HTTP endpoint and this endpoint is bound to a published Servlet.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-servlet</artifactId>
    <version>x.x.x</version>
    <\!-\- use the same version as your Camel core version \-->
</dependency>
```

Servlet is stream based, which means the input it receives is submitted to Camel as a stream. That means you will only be able to read the content of the stream once. If you find a situation where the message body appears to be empty or you need to access the data multiple times (eg: doing multicasting, or redelivery error handling) you should use Stream Caching or convert the message body to a String which is safe to be read multiple times.

3.40.1. URI format and options

```
servlet://relative_path[?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

Options

Name	Default Value	Description
httpBindingRef	null	Reference to an Camel `HttpBinding` object in the Registry. A `HttpBinding` implementation can be used to customize how to write a response.

Name	Default Value	Description
matchOnUriPrefix	false	Whether or not the CamelServlet should try to find a target consumer by matching the URI prefix, if no exact match is found.
servletName	CamelServlet	Specifies the servlet name that the servlet endpoint will bind to. This name should match the name you define in web.xml file.

3.40.2. Message Headers

Camel will apply the same Message Headers as the Section 3.20, "HTTP4" component.

Camel will also populate **all** request.parameter and request.headers. For example, if a client request has the URL, http://myserver/myserver?orderid=123 , the exchange will contain a header named orderid with the value 123.

3.40.3. Usage

You can only consume from endpoints generated by the Servlet component. Therefore, it should only be used as input into your Camel routes. To issue HTTP requests against other HTTP endpoints, use the HTTP4 Component.

3.40.4. Sample

In this sample, we define a route that exposes a HTTP service at http://localhost:8080/camel/ services/hello . First, you need to publish the CamelHttpTransportServlet through the normal Web Container, or OSGi Service. Use the Web.xml file to publish the CamelHttpTransportServlet as follows:

```
<web-app>

  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <display-name>Camel Http Transport Servlet</display-name>
    <servlet-class>
        org.apache.camel.component.servlet.CamelHttpTransportServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Then you can define your route as follows:

```
from("servlet:///hello?matchOnUriPrefix=true").process(new Processor() {
    public void process(Exchange exchange)
        throws Exception {
        String contentType =
            exchange.getIn().getHeader(Exchange.CONTENT_TYPE, String.class);
        String path =
            exchange.getIn().getHeader(Exchange.HTTP_PATH, String.class);
        assertEquals("Got a wrong content type", CONTENT_TYPE, contentType);
        // assert Camel http header
        String charsetEncoding = exchange.getIn()
            .getHeader(Exchange.HTTP_CHARACTER_ENCODING, String.class);
        assertEquals("Got a wrong charset name from the message header",
            "UTF-8", charsetEncoding);
        // assert exchange charset
        assertEquals("Got a wrong charset name from the exchange property",
            "UTF-8", exchange.getProperty(Exchange.CHARSET_NAME));
        exchange.getOut().setHeader(Exchange.CONTENT_TYPE, contentType +
            "; charset=UTF-8");
        exchange.getOut().setHeader("PATH", path);
        exchange.getOut().setBody("<b>Hello World</b>");
    }
});
```

Specify the relative path for camel-servlet endpoint

Since we are binding the Http transport with a published servlet, and we don't know the servlet's application context path, the `camel-servlet` endpoint uses the relative path to specify the endpoint's URL. A client can access the `camel-servlet` endpoint through the servlet publish address: (`"http://localhost:8080/camel/services"`) + `RELATIVE_PATH("/hello")`.

See the Camel Website for more examples of this component in use.

3.41. Shiro Security

The **shiro-security** component in Camel is a security focused component, based on the Apache Shiro security project.

Apache Shiro is a powerful and flexible open-source security framework that cleanly handles authentication, authorization, enterprise session management and cryptography. The objective of the Apache Shiro project is to provide the most robust and comprehensive application security framework available while also being very easy to understand and extremely simple to use.

This Camel shiro-security component allows authentication and authorization support to be applied to different segments of a Camel route.

Shiro security is applied on a route using a Camel Policy. A Policy in Camel utilizes a strategy pattern for applying interceptors on Camel Processors. It offering the ability to apply cross-cutting concerns (for example. security, transactions etc) on sections/segments of a Camel route.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
```

```
    <artifactId>camel-shiro</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.41.1. Shiro Security Basics

To employ Shiro security on a Camel route, a ShiroSecurityPolicy object must be instantiated with security configuration details (including users, passwords, roles etc). This object must then be applied to a Camel route. This ShiroSecurityPolicy Object may also be registered in the Camel registry (JNDI or ApplicationContextRegistry) and then utilized on other routes in the Camel Context.

Configuration details are provided to the ShiroSecurityPolicy using an Ini file (properties file) or an Ini object. The Ini file is a standard Shiro configuration file containing user/role details as shown below

```
[users]
# user 'ringo' with password 'starr' and the 'sec-level1' role
ringo = starr, sec-level1
george = harrison, sec-level2
john = lennon, sec-level3
paul = mccartney, sec-level3

[roles]
# 'sec-level3' role has all permissions, indicated by the
# wildcard '*'
sec-level3 = *

# The 'sec-level2' role can do anything with access of permission
# readonly (*) to help
sec-level2 = zone1:*

# The 'sec-level1' role can do anything with access of permission
# readonly
sec-level1 = zone1:readonly:*
```

3.41.2. Instantiating a ShiroSecurityPolicy Object

A ShiroSecurityPolicy object is instantiated as follows

```
private final String iniResourcePath = "classpath:shiro.ini";
      private final byte[] passPhrase = {
          (byte) 0x08, (byte) 0x09, (byte) 0x0A, (byte) 0x0B,
          (byte) 0x0C, (byte) 0x0D, (byte) 0x0E, (byte) 0x0F,
          (byte) 0x10, (byte) 0x11, (byte) 0x12, (byte) 0x13,
          (byte) 0x14, (byte) 0x15, (byte) 0x16, (byte) 0x17};
      List<permission> permissionsList = new ArrayList<permission>();
      Permission permission = new WildcardPermission("zone1:readwrite:*");
      permissionsList.add(permission);

      final ShiroSecurityPolicy securityPolicy =
          new ShiroSecurityPolicy(iniResourcePath, passPhrase, true,
              permissionsList);
```

3.41.3. ShiroSecurityPolicy Options

Name	Default Value	Type	Description
iniResourcePath or ini	none	Resource String or Ini Object	A mandatory Resource String for the iniResourcePath or an instance of an Ini object must be passed to the security policy. Resources can be acquired from the file system, classpath, or URLs when prefixed with "file:, classpath:, or url:" respectively. For e.g "classpath:shiro.ini"
passPhrase	An AES 128 based key	byte[]	A passPhrase to decrypt ShiroSecurityToken(s) sent along with Message Exchanges
alwaysReauth-enticate	true	boolean	Setting to ensure re-authentication on every individual request. If set to false, the user is authenticated and locked such than only requests from the same user going forward are authenticated.
permissionsList	none	List<Permission>	A List of permissions required in order for an authenticated user to be authorized to perform further action i.e continue further on the route. If no Permissions list is provided to the ShiroSecurityPolicy object, then authorization is deemed as not required
cipherService	AES	org.apache.shiro. crypto.CipherService	Shiro ships with AES & Blowfish based CipherServices. You may use one these or pass in your own Cipher implementation

3.41.4. Applying Shiro Authentication on a Camel Route

The ShiroSecurityPolicy, tests and permits incoming message exchanges containing a encrypted SecurityToken in the Message Header to proceed further following proper authentication. The SecurityToken object contains a Username/Password details that are used to determine where the user is a valid user.

```
protected RouteBuilder createRouteBuilder() throws Exception {
        final ShiroSecurityPolicy securityPolicy =
            new ShiroSecurityPolicy("classpath:shiro.ini", passPhrase);

        return new RouteBuilder() {
            public void configure() {
                onException(UnknownAccountException.class).
                    to("mock:authenticationException");
                onException(IncorrectCredentialsException.class).
                    to("mock:authenticationException");
                onException(LockedAccountException.class).
                    to("mock:authenticationException");
                onException(AuthenticationException.class).
                    to("mock:authenticationException");

                from("direct:secureEndpoint").
                    to("log:incoming payload").
```

```
                policy(securityPolicy).
                to("mock:success");
        }
    };
}
```

3.41.5. Applying Shiro Authorization on a Camel Route

Authorization can be applied on a Camel route by associating a Permissions List with the ShiroSecurityPolicy. The Permissions List specifies the permissions necessary for the user to proceed with the execution of the route segment. If the user does not have the proper permission set, the request is not authorized to continue any further.

```
protected RouteBuilder createRouteBuilder() throws Exception {
        final ShiroSecurityPolicy securityPolicy =
            new ShiroSecurityPolicy(
                "./src/test/resources/securityconfig.ini", passPhrase);

        return new RouteBuilder() {
            public void configure() {
                onException(UnknownAccountException.class).
                    to("mock:authenticationException");
                onException(IncorrectCredentialsException.class).
                    to("mock:authenticationException");
                onException(LockedAccountException.class).
                    to("mock:authenticationException");
                onException(AuthenticationException.class).
                    to("mock:authenticationException");

                from("direct:secureEndpoint").
                    to("log:incoming payload").
                    policy(securityPolicy).
                    to("mock:success");
            }
        };
    }
```

3.41.6. Creating a ShiroSecurityToken and injecting it into a Message Exchange

A ShiroSecurityToken object may be created and injected into a Message Exchange using a Shiro Processor called ShiroSecurityTokenInjector. An example of injecting a ShiroSecurityToken using a ShiroSecurityTokenInjector in the client is shown below

```
ShiroSecurityToken shiroSecurityToken =
     new ShiroSecurityToken("ringo", "starr");
   ShiroSecurityTokenInjector shiroSecurityTokenInjector =
     new ShiroSecurityTokenInjector(shiroSecurityToken,
     passPhrase);

   from("direct:client")
     .process(shiroSecurityTokenInjector)
     .to("direct:secureEndpoint");
```

3.41.7. Sending Messages to routes secured by a ShiroSecurityPolicy

Messages and Message Exchanges sent along the Camel route where the security policy is applied need to be accompanied by a SecurityToken in the Exchange Header. The SecurityToken is an encrypted object that holds a Username and Password. The SecurityToken is encrypted using AES 128 bit security by default and can be changed to any cipher of your choice.

Given below is an example of how a request may be sent using a ProducerTemplate in Camel along with a SecurityToken

```
@Test
   public void testSuccessfulShiroAuthenticationWithNoAuthorization()
            throws Exception {
      //Incorrect password
      ShiroSecurityToken shiroSecurityToken =
            new ShiroSecurityToken("ringo", "stirr");

      // TestShiroSecurityTokenInjector extends ShiroSecurityTokenInjector
      TestShiroSecurityTokenInjector shiroSecurityTokenInjector =
         new TestShiroSecurityTokenInjector(shiroSecurityToken, passPhrase);

      successEndpoint.expectedMessageCount(1);
      failureEndpoint.expectedMessageCount(0);

      template.send("direct:secureEndpoint", shiroSecurityTokenInjector);

      successEndpoint.assertIsSatisfied();
      failureEndpoint.assertIsSatisfied();
   }
```

3.42. SMPP

This component provides access to an SMSC (Short Message Service Center) over the SMPP protocol to send and receive SMS. The JSMPP is used.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-smpp</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

This component has log level **DEBUG**, which can be helpful in debugging problems. If you use log4j, you can add the following line to your configuration:

```
log4j.logger.org.apache.camel.component.smpp=DEBUG
```

3.42.1. URI Format

```
smpp://[username@]hostname[:port][?options]
smpps://[username@]hostname[:port][?options]
```

If no **username** is provided, then Camel will provide the default value `smppclient`. If no **port** number is provided, then Camel will provide the default value `2775`. If the protocol name is "smpps", camel-smpp with try to use SSLSocket to init a connection to the server.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.42.2. URI Options

Name	Default Value	Description
`password`	`password`	Specifies the password to use to log into the SMSC.
`systemType`	`cp`	This parameter is used to categorize the type of ESME (External Short Message Entity) that is binding to the SMSC (max. 13 characters).
`alphabet`	`0`	Defines encoding of data according the SMPP 3.4 specification, section 5.2.19. Example data encodings are: `0` : SMSC Default Alphabet 4 : 8 bit Alphabet 8 : UCS2 Alphabet
`encoding`	`ISO-8859-1`	Defines the encoding scheme of the short message user data. Only for SubmitSm, ReplaceSm and SubmitMulti.
`enquireLinkTimer`	`5000`	Defines the interval in milliseconds between the confidence checks. The confidence check is used to test the communication path between an ESME and an SMSC.
`transactionTimer`	`10000`	Defines the maximum period of inactivity allowed after a transaction, after which an SMPP entity may assume that the session is no longer active. This timer may be active on either communicating SMPP entity (that is, SMSC or ESME).
`initialReconnectDelay`	`5000`	Defines the initial delay in milliseconds after the consumer/producer tries to reconnect to the SMSC, after the connection was lost.
`reconnectDelay`	`5000`	Defines the interval in milliseconds between the reconnect attempts, if the connection to the SMSC was lost and the previous was not succeed.
`registeredDelivery`	`1`	Only for SubmitSm, ReplaceSm and SubmitMulti and DataSm. Is used to request an SMSC delivery receipt and/or SME originated acknowledgements. The following values are defined: `0` : No SMSC delivery receipt requested. `1` : SMSC delivery receipt requested where final delivery outcome is success or failure. `2` : SMSC delivery receipt requested where the final delivery outcome is delivery failure.
`serviceType`	`CMT`	The service type parameter can be used to indicate the SMS Application service associated with the message. The following generic service_types are defined: `CMT` : Cellular Messaging `CPT` : Cellular Paging `VMN` : Voice Mail Notification `VMA` : Voice Mail Alerting `WAP` : Wireless Application Protocol `USSD` : Unstructured Supplementary Services Data
`sourceAddr`	`1616`	Defines the address of SME (Short Message Entity) which originated this message.

Name	Default Value	Description
destAddr	1717	Only for SubmitSm, SubmitMulti, CancelSm and DataSm. Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
sourceAddrTon	0	Defines the type of number (TON) to be used in the SME originator address parameters. The following TON values are defined: 0 : Unknown 1 : International 2 : National 3 : Network Specific 4 : Subscriber Number 5 : Alphanumeric 6 : Abbreviated
destAddrTon	0	Only for SubmitSm, SubmitMulti, CancelSm and DataSm. Defines the type of number (TON) to be used in the SME destination address parameters. Same as the sourceAddrTon URI options listed above.
sourceAddrNpi	0	Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. The following NPI values are defined: 0 : Unknown 1 : ISDN (E163/E164) 2 : Data (X.121) 3 : Telex (F.69) 6 : Land Mobile (E.212) 8 : National 9 : Private 10 : ERMES 13 : Internet (IP) 18 : WAP Client Id (to be defined by WAP Forum)
destAddrNpi	0	Only for SubmitSm, SubmitMulti, CancelSm and DataSm. Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. Same as the sourceAddrNpi URI options listed above.
priorityFlag	1	Only for SubmitSm, SubmitMulti. Allows the originating SME to assign a priority level to the short message. Four Priority Levels are supported: 0 : Level 0 (lowest) priority 1 : Level 1 priority 2 : Level 2 priority 3 : Level 3 (highest) priority
replaceIfPresentFlag	0	Only for SubmitSm, SubmitMulti. Used to request the SMSC to replace a previously submitted message, that is still pending delivery. The SMSC will replace an existing message provided that the source address, destination address and service type match the same fields in the new message. The following replace if present flag values are defined: 0 : Don't replace 1 : Replace
typeOfNumber	0	Defines the type of number (TON) to be used in the SME. Same as the sourceAddrTon URI options listed above.
numberingPlanIndicator	0	Defines the numeric plan indicator (NPI) to be used in the SME. Same as the sourceAddrNpi URI options listed above.
lazySessionCreation	false	Sessions can be lazily created to avoid exceptions, if the SMSC is not available when the Camel producer is started.
httpProxyHost	null	If you need to tunnel SMPP through a HTTP proxy, set this attribute to the hostname or ip address of your HTTP proxy.
httpProxyPort	3128	If you need to tunnel SMPP through a HTTP proxy, set this attribute to the port of your HTTP proxy.
httpProxyUsername	null	If your HTTP proxy requires basic authentication, set this attribute to the username required for your HTTP proxy.
httpProxyPassword	null	If your HTTP proxy requires basic authentication, set this attribute to the password required for your HTTP proxy.

You can have as many of these options as you like, for example:

```
smpp://smppclient@localhost:2775?password=password&enquireLinkTimer=
3000&transactionTimer=5000&systemType=consumer
```

3.42.3. Producer Message Headers

The following message headers can be used to affect the behavior of the SMPP producer

Header	Type	Description
`CamelSmppDestAddr`	List/String	Only for SubmitSm, SubmitMulti, CancelSm and DataSm. Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.
`CamelSmppDestAddrTon`	Byte	Only for SubmitSm, SubmitMulti, CancelSm and DataSm. Defines the type of number (TON) to be used in the SME destination address parameters. Same as the `sourceAddrTon` URI options listed above.
`CamelSmppDestAddrNpi`	Byte	Only for SubmitSm, SubmitMulti, CancelSm and DataSm. Defines the numeric plan indicator (NPI) to be used in the SME destination address parameters. Same as the `sourceAddrNpi` URI options listed above.
`CamelSmppSourceAddr`	String	Defines the address of SME (Short Message Entity) which originated this message.
`CamelSmppSourceAddrTon`	Byte	Defines the type of number (TON) to be used in the SME originator address parameters. Same as the `sourceAddrTon` URI options listed above.
`CamelSmppSourceAddrNpi`	Byte	Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. Same as the `sourceAddrNpi` URI options listed above.
`CamelSmppServiceType`	String	The service type parameter can be used to indicate the SMS Application service associated with the message. Same as the `serviceType` URI options listed above.
`CamelSmppRegistered Delivery`	Byte	Only for SubmitSm, SubmitMulti, CancelSm and DataSm. Same as the `registeredDelivery` URI options listed above.
`CamelSmppPriorityFlag`	Byte	Only for SubmitSm and SubmitMulti. Same as the `priorityFlag` URI options listed above.
`CamelSmppSchedule DeliveryTime`	Date	Only for SubmitSm, SubmitMulti, ReplaceSm. This parameter specifies the scheduled time at which the message delivery should be first attempted. It defines either the absolute date and time or relative time from the current SMSC time at which delivery of this message will be attempted by the SMSC. It can be specified in either absolute time format or relative time format. The encoding of a time format is specified in Chapter 7.1.1. in the SMPP specification v3.4.
`CamelSmppValidityPeriod`	String/Date	Only for SubmitSm, SubmitMulti and ReplaceSm. The validity period parameter indicates the SMSC expiration time, after which the message should be discarded if not

Header	Type	Description
		delivered to the destination. It can be defined in absolute time format or relative time format. The encoding of absolute and relative time format is specified in chapter 7.1.1 in the smpp specification v3.4.
CamelSmppReplace IfPresentFlag	Byte	The replace if present flag parameter is used to request the SMSC to replace a previously submitted message, that is still pending delivery. The SMSC will replace an existing message provided that the source address, destination address and service type match the same fields in the new message. The following values are defined: 0 : Don't replace 1 : Replace
CamelSmppAlphabet	Byte	Only for SubmitSm, SubmitMulti and ReplaceSm. Same as the alphabet URI options listed above.

The following message headers are used by the SMPP producer to set the response from the SMSC in the message header

Header	Type	Description
CamelSmppId	String or List<String>	the id to identify the submitted short message for later use (delivery receipt, query sm, cancel sm, replace sm). In case of a ReplaceSm, QuerySm, CancelSm and DataSm this header value is a String. In case of a SubmitSm or SubmitMultiSm this header vaule is a List<String>.
CamelSmppSent MessageCount	Integer	For SubmitSm and SubmitMultiSm only - the total number of messages which has been sent.
CamelSmppError	Map<String, List<Map<String, Object>>>	For SubmitMultiSm only - The errors which occurred by sending the short message(s) the form Map<String, List<Map<String, Object>>>}} (messageID : (destAddr : address, error : errorCode)).

3.42.4. Consumer Message Headers

The following message headers are used by the SMPP consumer to set the request data from the SMSC in the message header

Header	Description
CamelSmppSequenceNumber	**only for alert notification, deliver sm and data sm** : A sequence number allows a response PDU to be correlated with a request PDU. The associated SMPP response PDU must preserve this field.
CamelSmppCommandId	**only for alert notification, deliver sm and data sm** : The command id field identifies the particular SMPP PDU. For the complete list of defined values see chapter 5.1.2.1 in the smpp specification v3.4.
CamelSmppSourceAddr	**only for alert notification, deliver sm and data sm** : Defines the address of SME (Short Message Entity) which originated this message.

Header	Description
CamelSmppSourceAddrNpi	**only for alert notification and data sm** : Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. Same as the `sourceAddrNpi` URI options listed above.
CamelSmppSourceAddrTon	**only for alert notification and data sm** : Defines the type of number (TON) to be used in the SME originator address parameters. Same as the `sourceAddrTon` URI options listed above.
CamelSmppEsmeAddr	**only for alert notification** : Defines the destination ESME address. For mobile terminated messages, this is the directory number of the recipient MS.
CamelSmppEsmeAddrNpi	**only for alert notification** : Defines the numeric plan indicator (NPI) to be used in the ESME originator address parameters. Same as the `sourceAddrNpi` URI options listed above.
CamelSmppEsmeAddrTon	**only for alert notification** : Defines the type of number (TON) to be used in the ESME originator address parameters. The following TON values are defined: Same as the `sourceAddrTon` URI options listed above.
CamelSmppId	**only for smsc delivery receipt and data sm** : The message ID allocated to the message by the SMSC when originally submitted.
CamelSmppDelivered	**only for smsc delivery receipt** : Number of short messages delivered. This is only relevant where the original message was submitted to a distribution list.The value is padded with leading zeros if necessary.
CamelSmppDoneDate	**only for smsc delivery receipt** : The time and date at which the short message reached its final state. The format is as follows: YYMMDDhhmm.
CamelSmppStatus	**only for smsc delivery receipt and data sm** : The final status of the message. The following values are defined: `DELIVRD` : Message is delivered to destination `EXPIRED` : Message validity period has expired. `DELETED` : Message has been deleted. `UNDELIV` : Message is undeliverable `ACCEPTD` : Message is in accepted state (that is, has been manually read on behalf of the subscriber by customer service) `UNKNOWN` : Message is in invalid state `REJECTD` : Message is in a rejected state
CamelSmppError	**only for smsc delivery receipt** : Where appropriate this may hold a Network specific error code or an SMSC error code for the attempted delivery of the message. These errors are Network or SMSC specific and are not included here.
CamelSmppSubmitDate	**only for smsc delivery receipt** : The time and date at which the short message was submitted. In the case of a message which has been replaced, this is the date that the original message was replaced. The format is as follows: YYMMDDhhmm.
CamelSmppSubmitted	**only for smsc delivery receipt** : Number of short messages originally submitted. This is only relevant when the original message was submitted to a distribution list.The value is padded with leading zeros if necessary.
CamelSmppDestAddr	**only for deliver sm and data sm** : Defines the destination SME address. For mobile terminated messages, this is the directory number of the recipient MS.

Header	Description
CamelSmppScheduleDeliveryTime	**only for deliver sm and data sm** : This parameter specifies the scheduled time at which the message delivery should be first attempted. It defines either the absolute date and time or relative time from the current SMSC time at which delivery of this message will be attempted by the SMSC. It can be specified in either absolute time format or relative time format. The encoding of a time format is specified in Section 7.1.1. in the smpp specification v3.4.
CamelSmppValidityPeriod	**only for deliver sm** : The validity period parameter indicates the SMSC expiration time, after which the message should be discarded if not delivered to the destination. It can be defined in absolute time format or relative time format. The encoding of absolute and relative time format is specified in Section 7.1.1 in the smpp specification v3.4.
CamelSmppServiceType	**only for deliver sm and data sm** : The service type parameter indicates the SMS Application service associated with the message.
CamelSmppRegisteredDelivery	Only for DataSm. Is used to request an delivery receipt and/or SME originated acknowledgements. Same as the `registeredDelivery` URI options listed above.
CamelSmppDestAddrNpi	Only for DataSm. Defines the numeric plan indicator (NPI) in the destination address parameters. Same as the `sourceAddrNpi` URI options listed above.
CamelSmppDestAddrTon	Only for DataSm. Defines the type of number (TON) in the destination address parameters. Same as the `sourceAddrTon` URI options listed above.
CamelSmppMessageType	Identifies the type of an incoming message: `AlertNotification` : an SMSC alert notification `DataSm` : an SMSC data short message `DeliveryReceipt` : an SMSC delivery receipt `DeliverSm` : an SMSC deliver short message

JSMPP library

See the documentation of the JSMPP Library for more details about the underlying library.

3.42.5. Samples

A route which sends an SMS using the Java DSL:

```
from("direct:start")
    .to("smpp://smppclient@localhost:2775?
        password=password&enquireLinkTimer=3000&transactionTimer=
            5000&systemType=producer");
```

A route which sends an SMS using the Spring XML DSL:

```
<route>
    <from uri="direct:start"/>
    <to uri="smpp://smppclient@localhost:2775?
        password=password&amp;enquireLinkTimer=3000&amp;transactionTimer=
```

```
        5000&amp;systemType=producer"/>
</route>
```

A route which receives an SMS using the Java DSL:

```
from("smpp://smppclient@localhost:2775?password=password&enquireLinkTimer=
    3000&transactionTimer=5000&systemType=consumer")
    .to("bean:foo");
```

A route which receives an SMS using the Spring XML DSL:

```
<route>
    <from uri="smpp://smppclient@localhost:2775?
        password=password&amp;enquireLinkTimer=3000&amp;
        transactionTimer=5000&amp;systemType=consumer"/>
    <to uri="bean:foo"/>
</route>
```

SMSC simulator

If you need an SMSC simulator for your test, you can use the simulator provided by Logica.

3.43. SNMP

The **snmp:** component gives you the ability to poll SNMP capable devices or receiving traps.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-snmp</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.43.1. URI format

```
snmp://hostname[:port][?Options]
```

The component supports polling OID values from an SNMP enabled device and receiving traps.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.43.2. Options

Name	Default Value	Description
type	none	The type of action you want to perform. You can enter here POLL or TRAP. The value POLL will instruct the

Name	Default Value	Description
		endpoint to poll a given host for the supplied OID keys. If you put in `TRAP` you will setup a listener for SNMP Trap Events.
`address`	none	This is the IP address and the port of the host to poll or where to setup the Trap Receiver. Example: `127.0.0.1:162`
`protocol`	`udp`	Here you can select which protocol to use. You can use either `udp` or `tcp`.
`retries`	`2`	Defines how often a retry is made before canceling the request.
`timeout`	`1500`	Sets the timeout value for the request in milliseconds.
`snmpVersion`	0 (which means SNMPv1)	Sets the snmp version for the request.
`snmpCommunity`	`public`	Sets the community octet string for the snmp request.
`delay`	60 seconds	Defines the delay in seconds between to poll cycles.
`oids`	none	Defines which values you are interested in. Please have a look at the Wikipedia to get a better understanding. You may provide a single OID or a comma separated list of OIDs. Example: oids="1.3.6.1.2.1.1.3.0 , 1.3.6.1.2.1.25.3.2.1.5.1 , 1.3.6.1.2 .1.25.3.5.1.1.1 , 1.3.6.1.2.1.43.5.1.1.11.1"

3.43.3. The result of a poll

Given the situation, that I poll for the following OIDs:

Example 3.1. oids

```
1.3.6.1.2.1.1.3.0
1.3.6.1.2.1.25.3.2.1.5.1
1.3.6.1.2.1.25.3.5.1.1.1
1.3.6.1.2.1.43.5.1.1.11.1
```

The result will be the following:

Example 3.2. Result of toString conversion

```
<?xml version="1.0" encoding="UTF-8"?>
<snmp>
  <entry>
    <oid>1.3.6.1.2.1.1.3.0</oid>
    <value>6 days, 21:14:28.00</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.25.3.2.1.5.1</oid>
    <value>2</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.25.3.5.1.1.1</oid>
    <value>3</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.43.5.1.1.11.1</oid>
    <value>6</value>
  </entry>
  <entry>
    <oid>1.3.6.1.2.1.1.1.0</oid>
    <value>My Very Special Printer Of Brand Unknown</value>
  </entry>
</snmp>
```

As you maybe recognized there is one more result than requested....1.3.6.1.2.1.1.1.0. This one is filled in by the device automatically in this special case. So it may absolutely happen, that you receive more than you requested...be prepared.

3.43.4. Examples

Polling a remote device:

```
snmp:192.168.178.23:161?protocol=udp&type=POLL&oids=1.3.6.1.2.1.1.5.0
```

Setting up a trap receiver (**Note that no OID info is needed here!**):

```
snmp:127.0.0.1:162?protocol=udp&type=TRAP
```

Routing example in Java: (converts the SNMP PDU to XML String)

```
from("snmp:192.168.178.23:161?protocol=udp&type=POLL"
   + "&oids=1.3.6.1.2.1.1.5.0").convertBodyTo(String.class).
   to("activemq:snmp.states");
```

3.44. Spring Integration

The **spring-integration:** component provides a bridge for Camel components to talk to Spring integration endpoints.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
```

```
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-integration</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.44.1. URI format

```
spring-integration:defaultChannelName[?options]
```

where **defaultChannelName** represents the default channel name which is used by the Spring Integration Spring context. It will equal to the `inputChannel` name for the Spring Integration consumer and the `outputChannel` name for the Spring Integration provider.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.44.2. Options

Name	Type	Description
`inputChannel`	String	The Spring integration input channel name that this endpoint wants to consume from, where the specified channel name is defined in the Spring context.
`outputChannel`	String	The Spring integration output channel name that is used to send messages to the Spring integration context.
`inOut`	String	The exchange pattern that the Spring integration endpoint should use. If `inOut=true` then a reply channel is expected, either from the Spring Integration Message header or configured on the endpoint.

3.44.3. Usage

The Spring integration component is a bridge that connects Camel endpoints with Spring integration endpoints through the Spring integration's input channels and output channels. Using this component, we can send Camel messages to Spring Integration endpoints or receive messages from Spring integration endpoints in a Camel routing context.

3.44.4. Examples

3.44.4.1. Using the Spring integration endpoint

You can set up a Spring integration endpoint using a URI, as follows:

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-integration.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

    <!-- Spring integration channels -->
    <channel id="inputChannel"/>
    <channel id="outputChannel"/>
    <channel id="onewayChannel"/>

    <!-- Spring integration service activators -->
    <service-activator input-channel="inputChannel" ref="helloService"
        method="sayHello"/>
    <service-activator input-channel="onewayChannel" ref="helloService"
        method="greet"/>

    <!-- custom bean -->
    <beans:bean id="helloService" class=
        "org.apache.camel.component.spring.integration.HelloWorldService"/>

    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="direct:twowayMessage"/>
            <to uri="spring-integration:inputChannel?inOut=true&
                inputChannel=outputChannel"/>
        </route>
        <route>
            <from uri="direct:onewayMessage"/>
            <to uri="spring-integration:onewayChannel?inOut=false"/>
        </route>
    </camelContext>


<!-- Spring integration channels -->
<channel id="requestChannel"/>
<channel id="responseChannel"/>

<!-- cusom Camel processor -->
<beans:bean id="myProcessor"
      class="org.apache.camel.component.spring.integration.MyProcessor"/>

<!-- Camel route -->
<camelContext  xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri=
         "spring-integration://requestChannel?outputChannel=responseChannel
&inOut=true"/>
        <process ref="myProcessor"/>
    </route>
</camelContext>
```

Or directly using a Spring integration channel name:

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-integration.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

   <!-- Spring integration channel -->
   <channel id="outputChannel"/>

   <camelContext xmlns="http://camel.apache.org/schema/spring">
      <route>
         <from uri="outputChannel"/>
         <to uri="mock:result"/>
      </route>
   </camelContext>
```

3.44.4.2. The Source and Target adapter

Spring integration also provides the Spring integration's source and target adapters, which can route messages from a Spring integration channel to a Camel endpoint or from a Camel endpoint to a Spring integration channel.

This example uses the following namespaces:

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
      xmlns:beans="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:camel-si="http://camel.apache.org/schema/spring/integration"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-integration.xsd
http://camel.apache.org/schema/spring/integration
http://camel.apache.org/schema/spring/integration/camel-spring-integration.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd
    ">
```

You can bind your source or target to a Camel endpoint as follows:

```
<!-- Create the Camel context here -->
<camelContext id="camelTargetContext"
      xmlns="http://camel.apache.org/schema/spring">
   <route>
      <from uri="direct:EndpointA" />
      <to uri="mock:result" />
   </route>
   <route>
      <from uri="direct:EndpointC"/>
      <process ref="myProcessor"/>
   </route>
```

```
</camelContext>

<!-- We can bind the camelTarget to the Camel context's endpoint by -->
<!-- specifying the camelEndpointUri attribute -->
<camel-si:camelTarget id="camelTargetA"
     camelEndpointUri="direct:EndpointA" expectReply="false">
   <camel-si:camelContextRef>
      camelTargetContext
   </camel-si:camelContextRef>
</camel-si:camelTarget>

<camel-si:camelTarget id="camelTargetB" camelEndpointUri="direct:EndpointC"
                  replyChannel="channelC" expectReply="true">
   <camel-si:camelContextRef>
      camelTargetContext
   </camel-si:camelContextRef>
</camel-si:camelTarget>

<camel-si:camelTarget id="camelTargetD" camelEndpointUri="direct:EndpointC"
                  expectReply="true">
   <camel-si:camelContextRef>
      camelTargetContext
   </camel-si:camelContextRef>
</camel-si:camelTarget>

<beans:bean id="myProcessor"
   class="org.apache.camel.component.spring.integration.MyProcessor"/>
```

3.45. Spring Security

The **camel-spring-security** component provides role-based authorization for Camel routes. It leverages the authentication and user services provided by Spring Security (formerly Acegi Security) and adds a declarative, role-based policy system to control whether a route can be executed by a given principal.

If you are not familiar with the Spring Security authentication and authorization system, please review the current reference documentation on the SpringSource web site linked above.

3.45.1. Creating authorization policies

Access to a route is controlled by an instance of a `SpringSecurityAuthorizationPolicy` object. A policy object contains the name of the Spring Security authority (role) required to run a set of endpoints and references to Spring Security `AuthenticationManager` and `AccessDecisionManager` objects used to determine whether the current principal has been assigned that role. Policy objects may be configured as Spring beans or by using an `<authorizationPolicy>` element in Spring XML.

The `<authorizationPolicy>` element may contain the following attributes:

Name	Default Value	Description
id	null	The unique Spring bean identifier which is used to reference the policy in routes (required)
access	null	The Spring Security authority name that is passed to the access decision manager (required)

Name	Default Value	Description
`authentication-Manager`	`authentication-Manager`	The name of the Spring Security `AuthenticationManager` object in the context
`accessDecision-Manager`	`accessDecision-Manager`	The name of the Spring Security `AccessDecisionManager` object in the context
`authentication-Adapter`	DefaultAuthentication-Adapter	The name of a **camel-spring-security** `AuthenticationAdapter` object in the context that is used to convert a `javax.security.auth.Subject` into a Spring Security `Authentication` instance.
`useThreadSecurity-Context`	`true`	If a `javax.security.auth.Subject` cannot be found in the In message header under Exchange.AUTHENTICATION, check the Spring Security `SecurityContextHolder` for an `Authentication` object.
`always-Reauthenticate`	`false`	If set to true, the `SpringSecurityAuthorizationPolicy` will always call `AuthenticationManager.authenticate()` each time the policy is accessed.

3.45.2. Controlling access to Camel routes

A Spring Security `AuthenticationManager` and `AccessDecisionManager` are required to use this component. Here is an example of how to configure these objects in Spring XML using the Spring Security namespace:

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:spring-security="http://www.springframework.org/schema/security"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security.xsd">
   <bean id="accessDecisionManager"
      class="org.springframework.security.access.vote.AffirmativeBased">
      <property name="allowIfAllAbstainDecisions" value="true"/>
      <property name="decisionVoters">
         <list>
            <bean class="org.springframework.security.access.vote.RoleVoter"/>
         </list>
      </property>
   </bean>

   <spring-security:authentication-manager alias="authenticationManager">
      <spring-security:authentication-provider
                  user-service-ref="userDetailsService"/>
   </spring-security:authentication-manager>

   <spring-security:user-service id="userDetailsService">
      <spring-security:user name="jim"
            password="jimspassword" authorities="ROLE_USER, ROLE_ADMIN"/>
```

```
        <spring-security:user name="bob"
                password="bobspassword" authorities="ROLE_USER"/>
    </spring-security:user-service>

</beans>
```

Now that the underlying security objects are set up, we can use them to configure an authorization policy and use that policy to control access to a route:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:spring-security="http://www.springframework.org/schema/security"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd
http://camel.apache.org/schema/spring-security
http://camel.apache.org/schema/spring-security/camel-spring-security.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.0.3.xsd">

    <!-- import the Spring security configuration  -->
    <import resource=
    "classpath:org/apache/camel/component/spring/security/commonSecurity.xml"/>

    <authorizationPolicy id="admin" access="ROLE_ADMIN"
            authenticationManager="authenticationManager"
            accessDecisionManager="accessDecisionManager"
            xmlns="http://camel.apache.org/schema/spring-security"/>

    <camelContext id="myCamelContext"
            xmlns="http://camel.apache.org/schema/spring">
            <route>
                <from uri="direct:start"/>
                <!-- The exchange should be authenticated with the role of  -->
                <!-- ADMIN before it is send to mock:endpoint -->
                <policy ref="admin">
                    <to uri="mock:end"/>
                </policy>
            </route>
    </camelContext>

</beans>
```

In this example, the endpoint mock:end will not be executed unless a Spring Security Authentication object that has been or can be authenticated and contains the ROLE_ADMIN authority can be located by the *admin* SpringSecurityAuthorizationPolicy.

3.45.3. Authentication

The process of obtaining security credentials that are used for authorization is not specified by this component. You can write your own processors or components which get authentication information from the exchange depending on your needs. For example, you might create a processor that gets credentials from an HTTP request header originating in the camel-jetty component. No matter how the credentials are collected, they need to be placed in the In message or the SecurityContextHolder so the **camel-spring-security** component can access them:

```
import javax.security.auth.Subject;
```

```
import org.apache.camel.*;
import org.apache.commons.codec.binary.Base64;
import org.springframework.security.authentication.*;


public class MyAuthService implements Processor {
    public void process(Exchange exchange) throws Exception {
        // get the username and password from the HTTP header
        // http://en.wikipedia.org/wiki/Basic_access_authentication

        String userpass = new String(Base64.decodeBase64(
                exchange.getIn().getHeader("Authorization", String.class)));
        String[] tokens= userpass.split(":");

        // create an Authentication object
        UsernamePasswordAuthenticationToken authToken =
                new UsernamePasswordAuthenticationToken(tokens[0], tokens[1]);

        // wrap it in a Subject
        Subject subject = new Subject();
        subject.getPrincipals().add(token);

        // place the Subject in the In message
        exchange.getIn().setHeader(Exchange.AUTHENTICATION, subject);

        // you could also do this if useThreadSecurityContext is set to true
        // SecurityContextHolder.getContext().setAuthentication(authToken);
    }
}
```

The `SpringSecurityAuthorizationPolicy` will automatically authenticate the `Authentication` object if necessary.

There are two issues to be aware of when using the `SecurityContextHolder` instead of or in addition to the `Exchange.AUTHENTICATION` header. First, the context holder uses a thread-local variable to hold the `Authentication` object. Any routes that cross thread boundaries, like **seda** or **jms**, will lose the `Authentication` object. Second, the Spring Security system appears to expect that an `Authentication` object in the context is already authenticated and has roles (see the Technical Overview for more details).

The default behavior of **camel-spring-security** is to look for a `Subject` in the `Exchange.AUTHENTICATION` header. This `Subject` must contain at least one principal, which must be a subclass of `org.springframework.security.core.Authentication`. You can customize the mapping of `Subject` to `Authentication` object by providing an implementation of the `org.apache.camel.component.spring.security.AuthenticationAdapter` to your `<authorizationPolicy>` bean. This can be useful if you are working with components that do not use Spring Security but do provide a `Subject`. At this time, only the camel-cxf component populates the `Exchange.AUTHENTICATION` header.

3.45.4. Handling authentication and authorization errors

If authentication or authorization fails in the `SpringSecurityAuthorizationPolicy`, a `CamelAuthorizationException` will be thrown. This can be handled using Camel's standard exception handling methods, like the Exception clause. The `CamelAuthorizationException` will have a reference

to the ID of the policy which threw the exception so you can handle errors based on the policy as well as the type of exception:

```
<onException>
  <exception>org.springframework.security.authentication.
      AccessDeniedException</exception>
  <choice>
    <when>
      <simple>${exception.policyId} == 'user'</simple>
      <transform>
        <constant>You do not have ROLE_USER access!</constant>
      </transform>
    </when>
    <when>
      <simple>${exception.policyId} == 'admin'</simple>
      <transform>
        <constant>You do not have ROLE_ADMIN access!</constant>
      </transform>
    </when>
  </choice>
</onException>
```

3.45.5. Dependencies

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-security</artifactId>
  <version>2.4.0</version>
</dependency>
```

This dependency will also pull in `org.springframework.security:spring-security-core:3.0.3.RELEASE` and `org.springframework.security:spring-security-config:3.0.3.RELEASE`.

3.46. SQL Component

The **sql:** component allows you to work with databases using JDBC queries. The difference between this component and Section 3.23, "JDBC" component is that in case of SQL the query is a property of the endpoint and it uses message payload as parameters passed to the query.

This component uses `spring-jdbc` behind the scenes for the SQL handling.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-sql</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
```

```
</dependency>
```

The SQL component also supports:

- a JDBC based repository for the Section 2.18, "Idempotent Consumer" EIP pattern. See further below.

- a JDBC based repository for the Aggregator EIP pattern. See further below.

3.46.1. URI format

⊗ Warning

The SQL component can only be used to define producer endpoints. In other words, you cannot define an SQL endpoint in a `from()` statement.

The SQL component uses the following endpoint URI notation:

```
sql:select * from table where id=# order by name[?options]
```

Notice that the standard ? symbol that denotes the parameters to an SQL query is substituted with the # symbol, because the ? symbol is used to specify options for the endpoint. The ? symbol replacement can be configured on endpoint basis.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.46.2. Options

Option	Type	Default	Description
dataSourceRef	String	null	Reference to a `DataSource` to look up in the registry.
placeholder	String	#	Specifies a character that will be replaced to ? in SQL query. Notice, that it is simple `String.replaceAll()` operation and no SQL parsing is involved (quoted strings will also change)
template.<xxx>		null	Sets additional options on the Spring `JdbcTemplate` that is used behind the scenes to execute the queries. For instance, `template.maxRows=10`. For detailed documentation, see the JdbcTemplate javadoc documentation.

3.46.3. Treatment of the message body

The SQL component tries to convert the message body to an object of `java.util.Iterator` type and then uses this iterator to fill the query parameters (where each query parameter is represented by a # symbol (or configured placeholder) in the endpoint URI). If the message body is not an array or collection, the conversion results in an iterator that iterates over only one object, which is the body itself.

For example, if the message body is an instance of `java.util.List`, the first item in the list is substituted into the first occurrence of # in the SQL query, the second item in the list is substituted into the second occurrence of #, and so on.

3.46.4. Result of the query

For `select` operations, the result is an instance of `List<Map<String, Object>>` type, as returned by the JdbcTemplate.queryForList() method. For `update` operations, the result is the number of updated rows, returned as an `Integer`.

3.46.5. Header values

When performing `update` operations, the SQL Component stores the update count in the following message headers:

Header	Description
`CamelSqlUpdateCount`	The number of rows updated for `update` operations, returned as an `Integer` object.
`CamelSqlRowCount`	The number of rows returned for `select` operations, returned as an `Integer` object.

3.46.6. Configuration in Camel

The SQL component must be configured before it can be used. In Spring, you can configure it as follows:

```
<bean id="sql" class="org.apache.camel.component.sql.SqlComponent">
    <property name="dataSource" ref="myDS"/>
</bean>

<bean id="myDS"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver" />
   <property name="url" value="jdbc:mysql://localhost:3306/ds" />
   <property name="username" value="username" />
   <property name="password" value="password" />
</bean>
```

You can now set a reference to a `DataSource` in the URI directly:

```
sql:select * from table where id=# order by name?dataSourceRef=myDS
```

3.46.8. Sample

In the sample below we execute a query and retrieve the result as a `List` of rows, where each row is a `Map<String, Object>` and the key is the column name.

First, we set up a table to use for our sample. As this is based on an unit test, we'll do it using java code:

```
// this is the database we create with some initial data for our unit test
jdbcTemplate.execute("create table projects (id integer primary key,"
                    + "project varchar(10), license varchar(5))");
```

```
jdbcTemplate.execute("insert into projects values (1, 'Camel', 'ASF')");
jdbcTemplate.execute("insert into projects values (2, 'AMQ', 'ASF')");
jdbcTemplate.execute("insert into projects values (3, 'Linux', 'XXX')");
```

Then we configure our route and our `sql` component. Notice that we use a `direct` endpoint in front of the `sql` endpoint. This allows us to send an exchange to the `direct` endpoint with the URI, `direct:simple`, which is much easier for the client to use than the long `sql:` URI. Note that the `DataSource` is looked up up in the registry, so we can use standard Spring XML to configure our `DataSource`.

```
   from("direct:simple")
   .to("sql:select * from projects where license=# order by id?dataSourceRef=
jdbc/myDataSource").to("mock:result");
```

And then we fire the message into the `direct` endpoint that will route it to our `sql` component that queries the database.

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedMessageCount(1);
// send the query to direct that will route it to the sql where we will
// execute the query and bind the parameters with the data from the body.
// The body only contains one value in this case (XXX) but if we should use
// multiple values then the body will be iterated so we could supply a
// List<String> instead containing each binding value.
template.sendBody("direct:simple", "XXX");

mock.assertIsSatisfied();

// the result is a List
List received = assertIsInstanceOf(
      List.class, mock.getReceivedExchanges().get(0).getIn().getBody());

// and each row in the list is a Map
Map row = assertIsInstanceOf(Map.class, received.get(0));

// and we should be able the get the project
// from the map that should be Linux
assertEquals("Linux", row.get("PROJECT"));
```

We could configure the `DataSource` in Spring XML as follows:

```
<jee:jndi-lookup id="myDS" jndi-name="jdbc/myDataSource"/>
```

3.46.9. Using the JDBC based idempotent repository

Available as of Camel 2.7 : In this section we will use the JDBC based idempotent repository.

First we need to setup a `javax.sql.DataSource` in the Spring XML file:

```
<bean id="dataSource"
   class="org.springframework.jdbc.datasource.SingleConnectionDataSource">
   <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
   <property name="url" value="jdbc:hsqldb:mem:camel_jdbc"/>
   <property name="username" value="sa"/>
   <property name="password" value=""/>
```

```
</bean>
```

And we can create our JDBC idempotent repository in the Spring XML file as well:

```
<bean id="messageIdRepository" class=
    "org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository">
    <constructor-arg ref="dataSource" />
    <constructor-arg value="myProcessorName" />
</bean>

<camel:camelContext>

    <camel:errorHandler
        id="deadLetterChannel" type="DeadLetterChannel"
            deadLetterUri="mock:error">
            <camel:redeliveryPolicy maximumRedeliveries="0"
                maximumRedeliveryDelay="0" logStackTrace="false" />
    </camel:errorHandler>

    <camel:route id="JdbcMessageIdRepositoryTest"
                errorHandlerRef="deadLetterChannel">
        <camel:from uri="direct:start" />
        <camel:idempotentConsumer
            messageIdRepositoryRef="messageIdRepository">
            <camel:header>messageId</camel:header>
            <camel:to uri="mock:result" />
        </camel:idempotentConsumer>
    </camel:route>
</camel:camelContext>
```

3.46.10. Using the JDBC based aggregation repository

Available as of Camel 2.6

⚠️ **Using JdbcAggregationRepository in Camel 2.6**

> In Camel 2.6, the JdbcAggregationRepository is provided in the `camel-jdbc-aggregator` component. From Camel 2.7 onwards, the `JdbcAggregationRepository` is provided in the `camel-sql` component.

`JdbcAggregationRepository` is an `AggregationRepository` which on the fly persists the aggregated messages. This ensures that you will not loose messages, as the default aggregator will use an in memory only `AggregationRepository`. The `JdbcAggregationRepository` allows together with Camel to provide persistent support for the Aggregator.

It has the following options:

Option	Type	Description
dataSource	DataSource	**Mandatory:** The `javax.sql.DataSource` to use for accessing the database.

Option	Type	Description
repositoryName	String	**Mandatory:** The name of the repository.
transactionManager	TransactionManager	**Mandatory:** The `org.springframework.transaction.PlatformTransactionManager` to mange transactions for the database. The TransactionManager must be able to support databases.
lobHandler	LobHandler	A `org.springframework.jdbc.support.lob.LobHandler` to handle Lob types in the database. Use this option to use a vendor specific LobHandler, for example when using Oracle.
returnOldExchange	boolean	Whether the get operation should return the old existing Exchange if any existed. By default this option is `false` to optimize as we do not need the old exchange when aggregating.
useRecovery	boolean	Whether or not recovery is enabled. This option is by default `true`. When enabled the Camel Aggregator automatic recover failed aggregated exchange and have them resubmitted.
recoveryInterval	long	If recovery is enabled then a background task is run every x'th time to scan for failed exchanges to recover and resubmit. By default this interval is 5000 milliseconds.
maximumRedeliveries	int	Allows you to limit the maximum number of redelivery attempts for a recovered exchange. If enabled then the Exchange will be moved to the dead letter channel if all redelivery attempts failed. By default this option is disabled. If this option is used then the `deadLetterUri` option must also be provided.
deadLetterUri	String	An endpoint uri for a Section 2.10, "Dead Letter Channel" where exhausted recovered Exchanges will be moved. If this option is used then the `maximumRedeliveries` option must also be provided.

3.46.10.1. What is preserved when persisting

`JdbcAggregationRepository` will only preserve any `Serializable` compatible data types. If a data type is not such a type it is dropped and a `WARN` is logged. And it only persists the `Message` body and the `Message` headers. The `Exchange` properties are **not** persisted.

3.46.10.2. Recovery

The `JdbcAggregationRepository` will by default recover any failed Exchange. It does this by having a background tasks that scans for failed Exchange s in the persistent store. You can use the `checkInterval` option to set how often this task runs. The recovery works as transactional which ensures that Camel will try to

_____

recover and redeliver the failed Exchange. Any Exchange which was found to be recovered will be restored from the persistent store and resubmitted and send out again.

The following headers is set when an Exchange is being recovered/redelivered:

Header	Type	Description
Exchange.REDELIVERED	Boolean	Is set to true to indicate the Exchange is being redelivered.
Exchange. REDELIVERY_COUNTER	Integer	The redelivery attempt, starting from 1.

Only when an Exchange has been successfully processed it will be marked as complete which happens when the confirm method is invoked on the AggregationRepository. This means if the same Exchange fails again it will be kept retried until it success.

You can use option maximumRedeliveries to limit the maximum number of redelivery attempts for a given recovered Exchange. You must also set the deadLetterUri option so Camel knows where to send the Exchange when the maximumRedeliveries was hit.

You can see some examples in the unit tests of camel-sql, for example this test.

3.46.10.3. Database

To be operational, each aggregator uses two table: the aggregation and completed one. By convention the completed has the same name as the aggregation one suffixed with "_COMPLETED". The name must be configured in the Spring bean with the RepositoryName property. In the following example aggregation will be used.

The table structure definition of both table are identical: in both case a String value is used as key (**id**) whereas a Blob contains the exchange serialized in byte array. However one difference should be remembered: the **id** field does not have the same content depending on the table. In the aggregation table **id** holds the correlation Id used by the component to aggregate the messages. In the completed table, **id** holds the id of the exchange stored in corresponding the blob field.

Here is the SQL query used to create the tables, just replace "aggregation" with your aggregator repository name.

```
CREATE TABLE aggregation (
    id varchar(255) NOT NULL,
    exchange blob NOT NULL,
    constraint aggregation_pk PRIMARY KEY (id)
);
CREATE TABLE aggregation_completed (
    id varchar(255) NOT NULL,
    exchange blob NOT NULL,
    constraint aggregation_completed_pk PRIMARY KEY (id)
);
```

3.46.10.4. Codec (Serialization)

Since they can contain any type of payload, Exchanges are not serializable by design. It is converted into a byte array to be stored in a database BLOB field. All those conversions are handled by the JdbcCodec class. One detail of the code requires your attention: the ClassLoadingAwareObjectInputStream.

The `ClassLoadingAwareObjectInputStream` has been reused from the Apache ActiveMQ project. It wraps an `ObjectInputStream` and use it with the `ContextClassLoader` rather than the `currentThread` one. The benefit is to be able to load classes exposed by other bundles. This allows the exchange body and headers to have custom types object references.

3.46.10.5. Transaction

A Spring `PlatformTransactionManager` is required to orchestrate transaction.

3.46.10.6. Service (Start/Stop)

The `start` method verify the connection of the database and the presence of the required tables. If anything is wrong it will fail during starting.

3.46.10.7. Aggregator configuration

Depending on the targeted environment, the aggregator might need some configuration. As you already know, each aggregator should have its own repository (with the corresponding pair of table created in the database) and a data source. If the default lobHandler is not adapted to your database system, it can be injected with the `lobHandler` property.

Here is the declaration for Oracle:

```
<bean id="lobHandler"
      class="org.springframework.jdbc.support.lob.OracleLobHandler">
   <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/>
</bean>

<bean id="nativeJdbcExtractor" class=
"org.springframework.jdbc.support.nativejdbc.CommonsDbcpNativeJdbcExtractor"/>

<bean id="repo" class=
   "org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository">
   <property name="transactionManager" ref="transactionManager"/>
   <property name="repositoryName" value="aggregation"/>
   <property name="dataSource" ref="dataSource"/>
   <!-- Only with Oracle, else use default -->
   <property name="lobHandler" ref="lobHandler"/>
</bean>
```

3.47. SSH

The SSH component enables access to SSH servers such that you can send an SSH command, and process the response. Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
```

```
    <artifactId>camel-ssh</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.47.1. URI Format

ssh:[username[:password]@]host[:port][?options]

3.47.2. Options

Name	Default Value	Description
host		Hostname of SSH Server
host		Hostname of SSH Server
port	22	SSH Server port
username		Username to authenticate with SSH Server
password		Password used for authenticating with SSH Server. Used if keyPairProvider is null.
keyPairProvider		Refers to a org.apache.sshd.common.KeyPairProvider to use for loading keys for authentication. If this option is used, then password is not used.
keyType	ssh-rsa	Refers to a key type to load from keyPairProvider. The key types can for example be "ssh-rsa" or "ssh-dss".
certFilename		File name of the keyPairProvider.
timeout	30000	Milliseconds to wait beforing timing out connection to SSH Server.
initialDelay	1000	Consumer only: Milliseconds before polling the SSH server starts.
delay	500	Consumer only: Milliseconds before the next poll of the SSH Server.
useFixedDelay	true	Consumer only: Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.
pollCommand		Consumer only: Command to send to SSH Server during each poll cycle. Used only when acting as Consumer.

3.48. Stub

The stub: component provides a simple way to stub out any physical endpoints for easy testing. Just add stub: in front of any endpoint URI in order to stub out the endpoint. This is useful in development where you might wish to try a route without needing to connect to a specific SMTP or HTTP endpoint.

Internally the Stub component creates VM endpoints. The main difference between Stub and VM is that VM will validate the URI and parameters you give it, so putting vm: in front of a typical URI with query arguments will usually fail. Stub won't though as it basically ignores all query parameters to let you quickly stub out one or more endpoints in your route temporarily.

3.48.1. URI Format

```
stub:someUri
```

Where someUri can be any URI with any query parameters.

3.48.2. Samples

Here are some samples:

```
stub:smtp://somehost.foo.com?user=whatnot&something=else
```

```
stub:http://somehost.bar.com/something
```

3.49. Test

Testing of distributed and asynchronous processing is notoriously difficult. The Section 3.32, "Mock", Section 3.49, "Test" and DataSet endpoints work great with the Camel Testing Framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's large range of Components together with the powerful Bean Integration.

The **test** component extends the Section 3.32, "Mock" component to support pulling messages from another endpoint on startup to set the expected message bodies on the underlying Section 3.32, "Mock" endpoint. That is, you use the test endpoint in a route and messages arriving on it will be implicitly compared to some expected messages extracted from some other location.

So you can use, for example, an expected set of message bodies as files. This will then set up a properly configured Section 3.32, "Mock" endpoint, which is only valid if the received messages match the number of expected messages and their message payloads are equal.

3.49.1. URI format

```
test:expectedMessagesEndpointUri
```

where **expectedMessagesEndpointUri** refers to some other Component URI that the expected message bodies are pulled from before starting the test.

3.49.2. Example

For example, you could write a test case as follows:

```
from("seda:someEndpoint").
  to("test:file://data/expectedOutput?noop=true");
```

If your test then invokes the MockEndpoint.assertIsSatisfied(camelContext) method, your test case will perform the necessary assertions.

Here is a real example test case using Mock and Spring along with its Spring XML.

To see how you can set other expectations on the test endpoint, see the Section 3.32, "Mock" component.

3.50. Timer

The **timer:** component is used to generate message exchanges when a timer fires You can only consume events from this endpoint.

3.50.1. URI format

```
timer:name[?options]
```

where `name` is the name of the `Timer` object, which is created and shared across endpoints. So if you use the same name for all your timer endpoints, only one `Timer` object and thread will be used.

You can append query options to the URI in the following format, `?option=value&option=value&...`

Note: The IN body of the generated exchange is `null`. So `exchange.getIn().getBody()` returns `null`.

Advanced Scheduler

See also the Section 3.35, "Quartz" component that supports much more advanced scheduling.

Specify time in human friendly format

You can specify the time in human friendly syntax.

3.50.2. Options

Name	Default Value	Description
time	null	A `java.util.Date` the **first** event should be generated. If using the URI, the pattern expected is: `yyyy-MM-dd HH:mm:ss` or `yyyy-MM-dd'T'HH:mm:ss`.
pattern	null	Allows you to specify a custom `Date` pattern to use for setting the time option using URI syntax.
period	1000	If greater than 0, generate periodic events every `period` milliseconds.
delay	0	The number of milliseconds to wait before the first event is generated. Should not be used in conjunction with the `time` option.
fixedRate	false	Events take place at approximately regular intervals, separated by the specified period.
daemon	true	Specifies whether or not the thread associated with the timer endpoint runs as a daemon.

3.50.3. Exchange Properties

When the timer is fired, it adds the following information as properties to the `Exchange` :

Name	Type	Description
`org.apache.camel.timer.name`	`String`	The value of the `name` option.
`org.apache.camel.timer.time`	`Date`	The value of the `time` option.
`org.apache.camel.timer.period`	`long`	The value of the `period` option.
`org.apache.camel.timer.` `firedTime`	`Date`	The time when the consumer fired.

3.50.4. Message Headers

When the timer is fired, it adds the following information as headers to the IN message

Name	Type	Description
`firedTime`	`java.util.Date`	The time when the consumer fired

3.50.5. Sample

To set up a route that generates an event every 60 seconds:

```
from("timer://foo?fixedRate=true&period=60000").
      to("bean:myBean?method=someMethodName");
```

The above route will generate an event and then invoke the `someMethodName` method on the bean called `myBean` in the Registry such as JNDI or Spring.

And the route in Spring DSL:

```
<route>
    <from uri="timer://foo?fixedRate=true&amp;period=60000"/>
    <to uri="bean:myBean?method=someMethodName"/>
  </route>
```

3.51. Velocity

The **velocity:** component allows you to process a message using an Apache Velocity template. This can be ideal when using Templating to generate responses for requests.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-velocity</artifactId>
    <version>x.x.x</version>
```

```
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.51.1. URI format

```
velocity:templateName[?options]
```

where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (for example: file://folder/myfile.vm").

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.51.2. Options

Option	Default	Description
loaderCache	true	Velocity based file loader cache.
contentCache	true	Cache for the resource content when it is loaded.
encoding	null	Character encoding of the resource content.
propertiesFile	null	The URI of the properties file which is used for VelocityEngine initialization.

3.51.3. Message Headers

The velocity component sets a couple headers on the message (you can't set these yourself and velocity component will not set these headers which will cause some side effect on the dynamic template support):

Header	Description
CamelVelocityResource	The resource as an `org.springframework.core.io.Resource` object.
CamelVelocityResourceUri	The **templateName** as a `String` object.

Headers set during the Velocity evaluation are returned to the message and added as headers. Then it's possible to return values from Velocity to the Message.

For example, to set the header value of `fruit` in the Velocity template `.tm` :

```
$in.setHeader('fruit', 'Apple')
```

The `fruit` header is now accessible from the `message.out.headers`.

3.51.4. Velocity Context

Camel will provide exchange information in the Velocity context (just a `Map`). The `Exchange` is transfered as:

key	value
exchange	The `Exchange` itself.
headers	The headers of the In message.
camelContext	The Camel Context intance.
request	The In message.
in	The In message.
body	The In message body.
out	The Out message (only for InOut message exchange pattern).
response	The Out message (only for InOut message exchange pattern).

3.51.5. Hot reloading

The Velocity template resource is, by default, hot reloadable for both file and classpath resources (expanded jar). If you set `contentCache=true`, Camel will only load the resource once, and thus hot reloading is not possible. This scenario can be used in production, when the resource never changes.

3.51.6. Dynamic templates

Camel provides two headers by which you can define a different resource location for a template or the template content itself. If any of these headers is set then Camel uses this over the endpoint configured resource. This allows you to provide a dynamic template at runtime.

Header	Type	Description
CamelVelocityResourceUri	String	A URI for the template resource to use instead of the endpoint configured.
CamelVelocityTemplate	String	The template to use instead of the endpoint configured.

3.51.7. Samples

For example you could use something like

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

To use a Velocity template to formulate a response to a message for InOut message exchanges (where there is a `JMSReplyTo` header).

If you want to use InOnly and consume the message and send it to another destination, you could use the following route:

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

And to use the content cache, for example, for use in production, where the `.vm` template never changes:

```
from("activemq:My.Queue")
```

```
.to("velocity:com/acme/MyResponse.vm?contentCache=true")
.to("activemq:Another.Queue");
```

And a file based resource:

```
from("activemq:My.Queue")
   .to("velocity:file://myfolder/MyResponse.vm?contentCache=true")
   .to("activemq:Another.Queue");
```

In it is possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in")
   .setHeader("CamelVelocityResourceUri")
   .constant("path/to/my/template.vm")
   .to("velocity:dummy");
```

In it is possible to specify a template directly as a header the component should use dynamically via a header, so for example:

```
from("direct:in")
   .setHeader("CamelVelocityTemplate")
   .constant("Hi this is a velocity template" +
   "that can do templating ${body}")
   .to("velocity:dummy");
```

3.51.8. The Email Sample

In this sample we want to use Velocity templating for an order confirmation email. The email template is laid out in Velocity as:

```
Dear ${headers.lastName}, ${headers.firstName}

Thanks for the order of ${headers.item}.

Regards Studio Bookstore
${body}
```

And the java code:

```
private Exchange createLetter() {
   Exchange exchange = context.getEndpoint("direct:a").createExchange();
   Message msg = exchange.getIn();
   msg.setHeader("firstName", "Jens");
   msg.setHeader("lastName", "Hansen");
   msg.setHeader("item", "Factory Patterns");
   msg.setBody("PS: Next beer is on me, Joe");
   return exchange;
}

@Test
public void testVelocityLetter() throws Exception {
   MockEndpoint mock = getMockEndpoint("mock:result");
   mock.expectedMessageCount(1);
   mock.expectedBodiesReceived(
   "Dear Hansen, Jens\n\nThanks for the order of Factory Patterns.\n\n" +
```

```
    "Regards Studio Bookstore\nPS: Next beer is on me, Joe");

    template.send("direct:a", createLetter());

    mock.assertIsSatisfied();
}

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:a").
            to("velocity:org/apache/camel/component/velocity/letter.vm").
            to("mock:result");
        }
    };
}
```

3.52. VM

The **vm:** component provides asynchronous SEDA behavior so that messages are exchanged on a BlockingQueue and consumers are invoked in a separate thread pool to the producer.

This component differs from the Section 3.39, "SEDA" component in that VM supports communication across CamelContext instances, so you can use this mechanism to communicate across web applications, provided that the `camel-core.jar` is on the `system/boot` classpath.

This component is an extension to the Section 3.39, "SEDA" component.

3.52.1. URI format

```
vm:someName[?options]
```

where **someName** can be any string to uniquely identify the endpoint within the JVM (or at least within the classloader which loaded the camel-core.jar)

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.52.2. Options

See the Section 3.39, "SEDA" component for options and other important usage as the same rules apply for this Section 3.52, "VM" component.

3.52.3. Samples

In the route below we send the exchange to the VM queue that is working across CamelContext instances:

```
from("direct:in").bean(MyOrderBean.class).to("vm:order.email");
```

And then in another Camel context such as deployed as in another `.war` application:

```
from("vm:order.email").bean(MyOrderEmailSender.class);
```

3.53. XQuery Endpoint

The **xquery:** component allows you to process a message using an XQuery template. This can be ideal when using Templating to generate responses for requests.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-saxon</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.53.1. URI format

```
xquery:templateName
```

where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template.

For example you could use something like this:

```
from("activemq:My.Queue").
  to("xquery:com/acme/mytransform.xquery");
```

To use an XQuery template to formulate a response to a message for InOut message exchanges (where there is a `JMSReplyTo` header).

If you want to use InOnly, consume the message, and send it to another destination, you could use the following route:

```
from("activemq:My.Queue")
   .to("xquery:com/acme/mytransform.xquery")
   .to("activemq:Another.Queue");
```

3.54. XSLT

The **xslt:** component allows you to process a message using an XSLT template. This can be ideal when using Templating to generate responses for requests.

3.54.1. URI format

```
xslt:templateName[?options]
```

where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template. Refer to the Spring Documentation for more detail of the URI syntax

You can append query options to the URI in the following format, `?option=value&option=value&...`

Here are some example URIs

URI	Description
`xslt:com/acme/mytransform.xsl`	refers to the file com/acme/mytransform.xsl on the classpath
`xslt:file:///foo/bar.xsl`	refers to the file /foo/bar.xsl
`xslt:http://acme.com/cheese/foo.xsl`	refers to the remote http resource

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

3.54.2. Options

Name	Default Value	Description
`converter`	`null`	Option to override default XmlConverter. This will lookup for the converter in the Registry. The provided converted must be of type org.apache.camel.converter.jaxp.XmlConverter.
`transformerFactory`	`null`	**Camel 1.6** Option to override default TransformerFactory. This will lookup for the transformerFactory in the Registry. The provided transformer factory must be of type javax.xml.transform.TransformerFactory.
`transformerFactoryClass`	`null`	**Camel 1.6** Option to override default TransformerFactory. This will create a TransformerFactoryClass instance and set it to the converter.
`uriResolver`	`null`	**Camel 2.3** : Allows you to use a custom `javax.xml.transformation.URIResolver`. Camel will by default use its own implementation `org.apache.camel.builder.xml.XsltUriResolver` which is capable of loading from classpath.
`resultHandlerFactory`	`null`	**Camel 2.3:** Allows you to use a custom `org.apache.camel.builder.xml.ResultHandlerFactory` which is capable of using custom `org.apache.camel.builder.xml.ResultHandler` types.
`failOnNullBody`	`true`	**Camel 2.3:** Whether or not to throw an exception if the input body is null.

Name	Default Value	Description
deleteOutputFile	false	**Camel 2.6:** If you have `output=file` then this option dictates whether or not the output file should be deleted when the Exchange is done processing. For example suppose the output file is a temporary file, then it can be a good idea to delete it after use.
output	string	**Camel 2.3:** Option to specify which output type to use. Possible values are: `string`, `bytes`, `DOM`, `file`. The first three options are all in memory based, where as `file` is streamed directly to a `java.io.File`. For `file` you **must** specify the filename in the IN header with the key `Exchange.XSLT_FILE_NAME` which is also `CamelXsltFileName`. Also any paths leading to the filename must be created beforehand, otherwise an exception is thrown at runtime.
contentCache	true	**Camel 2.6:** Cache for the resource content (the stylesheet file) when it is loaded. If set to `false` Camel will reloader the stylesheet file on each message processing. This is good for development.

3.54.3. Using XSLT endpoints

For example you could use something like

```
from("activemq:My.Queue")
    .to("xslt:com/acme/mytransform.xsl");
```

To use an XSLT template to formulate a response for a message for InOut message exchanges (where there is a `JMSReplyTo` header).

If you want to use InOnly and consume the message and send it to another destination you could use the following route:

```
from("activemq:My.Queue")
    .to("xslt:com/acme/mytransform.xsl")
    .to("activemq:Another.Queue");
```

3.54.4. Getting Parameters into the XSLT to work with

By default, all headers are added as parameters which are available in the XSLT. To do this you will need to declare the parameter so it is then *useable*.

```
<setHeader headerName="myParam"><constant>42</constant></setHeader>
<to uri="xslt:MyTransform.xsl"/>
```

And the XSLT just needs to declare it at the top level for it to be available:

```
<xsl: ...... >

    <xsl:param name="myParam"/>
```

```
<xsl:template ...>
```

3.54.5. Spring XML versions

To use the above examples in Spring XML you would use something like

```
<camelContext xmlns="http://activemq.apache.org/camel/schema/spring">
    <route>
        <from uri="activemq:My.Queue"/>
        <to uri="xslt:org/apache/camel/spring/processor/example.xsl"/>
        <to uri="activemq:Another.Queue"/>
    </route>
 </camelContext>
```

There is a test case along with its Spring XML if you want a concrete example.

3.54.6. Using xsl:include

Camel provides its own implementation of `URIResolver` which allows Camel to load included files from the classpath and more intelligent than before.

For example this include:

```
<xsl:include href="staff_template.xsl"/>
```

This will now be located relative from the starting endpoint, which for example could be:

```
.to("xslt:org/apache/camel/component/xslt/staff_include_relative.xsl")
```

Which means Camel will locate the file in the **classpath** as `org/apache/camel/component/xslt/` `staff_template.xsl`. This allows you to use xsl include and have xsl files located in the same folder such as we do in the example `org/apache/camel/component/xslt`.

You can use the following two prefixes `classpath:` or `file:` to instruct Camel to look either in classpath or file system. If you omit the prefix then Camel uses the prefix from the endpoint configuration. If that neither has one, then classpath is assumed.

You can also refer back in the paths such as

```
<xsl:include href="../staff_other_template.xsl"/>
```

Which then will resolve the xsl file under `org/apache/camel/component`.

Chapter 4. Talend ESB Mediation Examples

The samples folder of the Talend ESB download contain examples that are provided by the Apache Camel project, as well as Talend ESB-specific examples showing multiple usages of Camel routing. Each Talend ESB sample has its own README file providing a full description of the sample along with deployment information using embedded Jetty or Talend OSGi container. The examples provided by the Apache Camel project and bundled with the Talend ESB are listed and explained on the Camel website; the below listing provides a summary of additional mediation examples provided in the Talend ESB distribution.

Example	Description
blueprint	Provides an example of deploying Camel routes as an OSGi bundle in the TESB container.
claimcheck	EAI patterns example demonstrating use of the Claim Check, Splitter, Reswquencer and Delayer patterns.
jaxrs-jms-http	Shows how a JAX-RS service can be offered an used with Camel transports.
jaxws-jms	Shows how to publish and call a CXF service using SOAP/JMS using Camel as a CXF transport.
spring-security	Example shows how to leverage Spring Security to secure Camel routes in general and also specifically when combined with CXF JAX-WS and JAX-RS endpoints.