



Talend ESB Mediation

Developer Guide

6.4.1

Adapted for v6.4.1. Supersedes previous releases.

Publication date: June 29, 2017

Copyright © 2017 Talend Inc. All rights reserved.

Copyright

This documentation is provided under the terms of the Creative Commons Public License (CCPL). For more information about what you can and cannot do with this documentation in accordance with the CCPL, please read: <http://creativecommons.org/licenses/by-nc-sa/2.0/>

This document may include documentation produced at The Apache Software Foundation which is licensed under The Apache License 2.0.

Notices

Talend and Talend ESB are trademarks of Talend, Inc.

Apache CXF, CXF, Apache Karaf, Karaf, Apache Cellar, Cellar, Apache Camel, Camel, Apache Maven, Maven, Apache Archiva, Archiva, Apache Syncope, Syncope, Apache ActiveMQ, ActiveMQ, Apache Log4j, Log4j, Apache Felix, Felix, Apache ServiceMix, ServiceMix, Apache Ant, Ant, Apache Derby, Derby, Apache Tomcat, Tomcat, Apache ZooKeeper, ZooKeeper, Apache Jackrabbit, Jackrabbit, Apache Santuario, Santuario, Apache DS, DS, Apache Avro, Avro, Apache Abdera, Abdera, Apache Chemistry, Chemistry, Apache CouchDB, CouchDB, Apache Kafka, Kafka, Apache Lucene, Lucene, Apache MINA, MINA, Apache Velocity, Velocity, Apache FOP, FOP, Apache HBase, HBase, Apache Hadoop, Hadoop, Apache Shiro, Shiro, Apache Axiom, Axiom, Apache Neethi, Neethi, Apache WSS4J, WSS4J are trademarks of The Apache Foundation. Eclipse Equinox is a trademark of the Eclipse Foundation, Inc. SoapUI is a trademark of SmartBear Software. Hyperic is a trademark of VMware, Inc. Nagios is a trademark of Nagios Enterprises, LLC.

All other brands, product names, company names, trademarks and service marks are the properties of their respective owners.

This product includes software developed at AOP Alliance (Java/J2EE AOP standards), ASM, AntLR, Apache ActiveMQ, Apache Ant, Apache Avro, Apache Axiom, Apache Axis, Apache Axis 2, Apache Batik, Apache CXF, Apache Camel, Apache Chemistry, Apache Common Http Client, Apache Common Http Core, Apache Commons, Apache Commons Bcel, Apache Commons JXPath, Apache Commons Lang, Apache Derby Database Engine and Embedded JDBC Driver, Apache Geronimo, Apache Hadoop, Apache Hive, Apache HttpClient, Apache HttpComponents Client, Apache JAMES, Apache Log4j, Apache Lucene Core, Apache Neethi, Apache POI, Apache Pig, Apache Qpid-Jms, Apache Tomcat, Apache Velocity, Apache WSS4J, Apache WebServices Common Utilities, Apache Xml-RPC, Apache Zookeeper, Box Java SDK (V2), CSV Tools, DataStax Java Driver for Apache Cassandra, Ehcache, Ezmorph, Ganymed SSH-2 for Java, Google APIs Client Library for Java, Google Gson, Groovy, Guava: Google Core Libraries for Java, H2 Embedded Database and JDBC Driver, HsqlDB, Ini4j, JClouds, JLine, JSON, JSR 305: Annotations for Software Defect Detection in Java, JUnit, Jackson Java JSON-processor, Java API for RESTful Services, Jaxb, Jaxen, Jettison, Jetty, Joda-Time, Json Simple, MetaStuff, Mondrian, OpenSAML, Paracel JDBC Driver, PostgreSQL JDBC Driver, Resty: A simple HTTP REST client for Java, Rocoto, SL4J: Simple Logging Facade for Java, SQLite JDBC Driver, Simple API for CSS, SshJ, StAX API, StAXON - JSON via StAX, Talend Camel Dependencies (Talend), The Castor Project, The Legion of the Bouncy Castle, W3C, Woden, Woodstox : High-performance XML processor, XML Pull Parser (XPP), Xalan-J, Xerces2, XmlBeans, XmlSchema Core, Xmlsec - Apache Santuario, Zip4J, atinject, dropbox-sdk-java: Java library for the Dropbox Core API, google-guice. Licensed under their respective license.

Table of Contents

Chapter 1. Introduction	1	
Chapter 2. Enterprise Integration Patterns	3	
2.1. List of EIPs	4	
2.2. Aggregator	8	
2.3. Claim Check	12	
2.4. Competing Consumers	14	
2.5. Composed Message Processor	15	
2.6. Content Based Router	15	
2.7. Content Enricher	16	
2.8. Content Filter	21	
2.9. Control Bus	22	
2.10. Correlation Identifier	22	
2.11. Dead Letter Channel	23	
2.12. Delayer	28	
2.13. Detour	30	
2.14. Durable Subscriber	31	
2.15. Dynamic Router	32	
2.16. Event Driven Consumer	34	
2.17. Event Message	35	
2.18. Guaranteed Delivery	36	
2.19. Idempotent Consumer	36	
2.20. Load Balancer	38	
2.21. Log	42	
2.22. Loop	44	
2.23. Message	45	
2.24. Message Bus	45	
2.25. Message Channel	46	
2.26. Message Dispatcher	46	
2.27. Message Endpoint	47	
2.28. Message Filter	47	
2.29. Message History	49	
2.30. Message Router	51	
2.31. Message Translator	52	
2.32. Messaging Gateway	53	
2.33. Messaging Mapper	54	
2.34. Multicast	54	
2.35. Normalizer	57	
2.36. Pipes and Filters	58	
2.37. Point to Point Channel	59	
2.38. Polling Consumer	59	
2.39. Publish Subscribe Channel	64	
2.40. Recipient List	65	
2.41. Request Reply	68	
2.42. Resequencer	69	
2.43. Return Address	72	
2.44. Routing Slip	73	
2.45. Sampling	76	
2.46. Scatter-Gather	77	
2.47. Selective Consumer	80	
2.48. Service Activator	80	
2.49. Sort	81	
2.50. Splitter	82	
2.51. Throttler	89	
2.52. Transactional Client	89	
2.53. Validate	94	
2.54. Wire Tap	95	
Chapter 3. Components	97	
3.1. ActiveMQ	103	
3.2. AHC	107	
3.3. Atom	113	
3.4. Apns	114	
3.5. Avro	118	
3.6. Bean	121	
3.7. Cache	123	
3.8. Class	130	
3.9. CMIS	131	
3.10. Context	133	
3.11. CouchDB	135	
3.12. Crypto (Digital Signatures)	136	
3.13. CXF	138	
3.14. CXF Bean Component	153	
3.15. CXFRS	156	
3.16. Direct	158	
3.17. Disruptor	160	
3.18. Elasticsearch	164	
3.19. Exec	165	
3.20. Facebook	168	
3.21. File	180	
3.22. Flatpack	197	
3.23. FOP	200	
3.24. Freemarker	202	
3.25. FTP	204	
3.26. Geocoder	213	
3.27. Guava EventBus	215	
3.28. HBase	217	
3.29. HDFS	225	
3.30. HDFS2	228	
3.31. HI7	231	
3.32. HTTP4	236	
3.33. Infinispan	246	
3.34. Jasypt	248	
3.35. JCR	251	
3.36. JDBC	253	
3.37. Jetty	256	
3.38. JGroups	263	
3.39. JMS	266	
3.40. JMX	278	
3.41. JPA	280	
3.42. Kafka	284	
3.43. Krati	286	
3.44. Jsch	289	
3.45. LDAP	290	
3.46. Log	292	
3.47. Lucene	296	
3.48. Mail	300	
3.49. MINA 2	305	
3.50. Mock	309	
3.51. MongoDB	314	
3.52. MQTT	325	
3.53. Mustache	327	
3.54. MyBatis	329	
3.55. Netty HTTP	331	
3.56. OptaPlanner	338	
3.57. Properties	340	
3.58. Quartz	348	
3.59. Quartz2	351	
3.60. RabbitMQ Component	355	
3.61. Ref	358	
3.62. RMI	359	
3.63. RSS	360	
3.64. Salesforce	362	
3.65. SAP NetWeaver	365	
3.66. SEDA	367	
3.67. Servlet	370	
3.68. Shiro Security	372	
3.69. SJMS	377	
3.70. SMPP	384	
3.71. SNMP	391	
3.72. Solr	393	
3.73. Splunk	395	
3.74. Spring Batch	398	
3.75. Spring Event	401	
3.76. Spring Integration	401	
3.77. Spring LDAP	405	
3.78. Spring Redis	406	
3.79. Spring Web Services	411	
3.80. Spring Security	419	
3.81. SQL Component	423	
3.82. SSH	434	
3.83. StAX	435	

3.84. Stomp	437
3.85. Stub	439
3.86. Test	439
3.87. Timer	440
3.88. Twitter	442
3.89. Velocity	445
3.90. Vertx	447
3.91. VM	448
3.92. Weather	449
3.93. Websocket	451
3.94. XQuery Endpoint	454
3.95. XSLT	455
3.96. Yammer	458
3.97. Zookeeper	463
Chapter 4. Talend ESB Mediation	
Examples	467



Chapter 1. Introduction



This manual covers the Apache Camel 2.10.x series.

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.



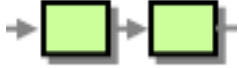
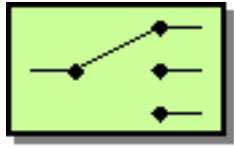
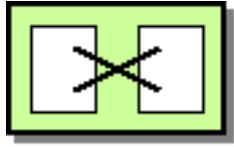
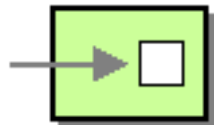


Chapter 2. Enterprise Integration Patterns



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


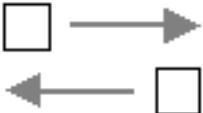
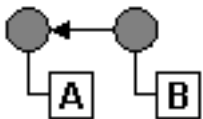

	<i>Message Channel</i>	How does one application communicate with another using messaging?
	<i>Message</i>	How can two applications connected by a message channel exchange a piece of information?
	<i>Pipes and Filters</i>	How can we perform complex processing on a message while maintaining independence and flexibility?
	<i>Message Router</i>	How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?
	<i>Message Translator</i>	How can systems using different data formats communicate with each other using messaging?
	<i>Message Endpoint</i>	How does an application connect to a messaging channel to send and receive messages?

2.1.2. Messaging Channels

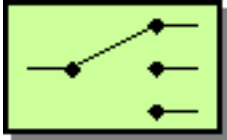
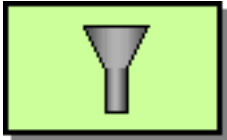
	<i>Point to Point Channel</i>	How can the caller be sure that exactly one receiver will receive the document or perform the call?
	<i>Publish Subscribe Channel</i>	How can the sender broadcast an event to all interested receivers?

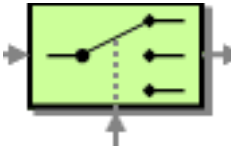
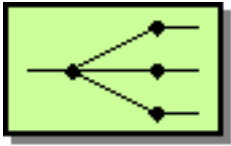
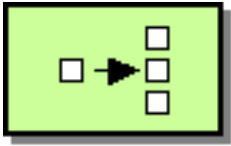
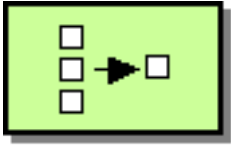
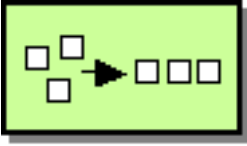
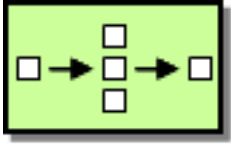
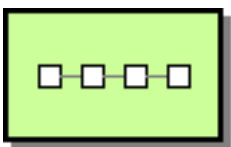
	<i>Dead Letter Channel</i>	What will the messaging system do with a message it cannot deliver?
	<i>Guaranteed Delivery</i>	How can the sender make sure that a message will be delivered, even if the messaging system fails?
	<i>Message Bus</i>	What is an architecture that enables separate applications 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

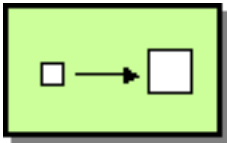
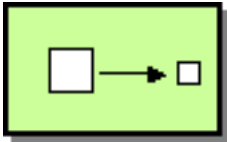
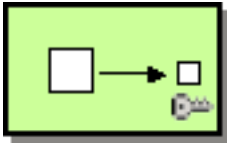
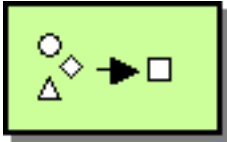
	<i>Event Message</i>	How can messaging be used to transmit events from one application to another?
	<i>Request Reply</i>	When an application sends a message, how can it get a response from the receiver?
	<i>Correlation Identifier</i>	How does a requestor that has received a reply know which request this is the reply for?
	<i>Return Address</i>	How does a replier know where to send the reply?

2.1.4. Message Routing

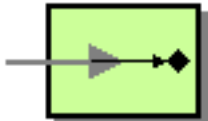



	<i>Content Based Router</i>	How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?
	<i>Message Filter</i>	How can a component avoid receiving uninteresting messages?

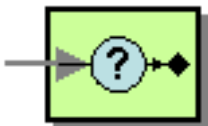
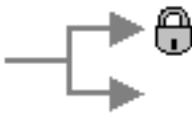
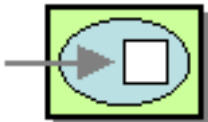
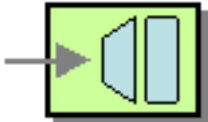
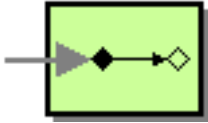

	<i>Dynamic Router</i>	How can you avoid the dependency of the router on all possible destinations while maintaining its efficiency?
	<i>Recipient List</i>	How do we route a message to a list of (static or dynamically) specified recipients?
	<i>Splitter</i>	How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?
	<i>Aggregator</i>	How do we combine the results of individual, but related messages so that they can be processed as a whole?
	<i>Resequencer</i>	How can we get a stream of related but out-of-sequence messages back into the correct order?
	<i>Composed Message Processor</i>	How can you maintain the overall message flow when processing a message consisting of multiple elements, each of which may require different processing?
	<i>Scatter-Gather</i>	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?
	<i>Routing Slip</i>	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?
	<i>Throttler</i>	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?
	<i>Sampling</i>	How can I sample one message out of many in a given period to avoid downstream route does not get overloaded?
	<i>Delayer</i>	How can I delay the sending of a message?
	<i>Load Balancer</i>	How can I balance load across a number of endpoints?
	<i>Multicast</i>	How can I route a message to a number of endpoints at the same time?
	<i>Loop</i>	How can I repeat processing a message in a loop?

2.1.5. Message Transformation


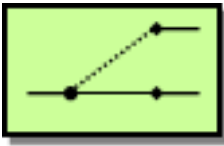
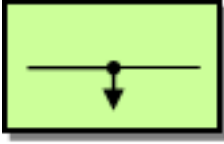
	<i>Content Enricher</i>	How do we communicate with another system if the message originator does not have all the required data items available?
	<i>Content Filter</i>	How do you simplify dealing with a large message, when you are interested only in a few data items?
	<i>Claim Check</i>	How can we reduce the data volume of message sent across the system without sacrificing information content?
	<i>Normalizer</i>	How do you process messages that are semantically equivalent, but arrive in a different format?
	<i>Sort</i>	How can I sort the body of a message?
	<i>Validate</i>	How can I validate a message?

2.1.6. Messaging Endpoints

	<i>Messaging Mapper</i>	How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?
	<i>Event Driven Consumer</i>	How can an application automatically consume messages as they become available?
	<i>Polling Consumer</i>	How can an application consume a message when the application is ready?
	<i>Competing Consumers</i>	How can a messaging client process multiple messages concurrently?
	<i>Message Dispatcher</i>	How can multiple consumers on a single channel coordinate their message processing?

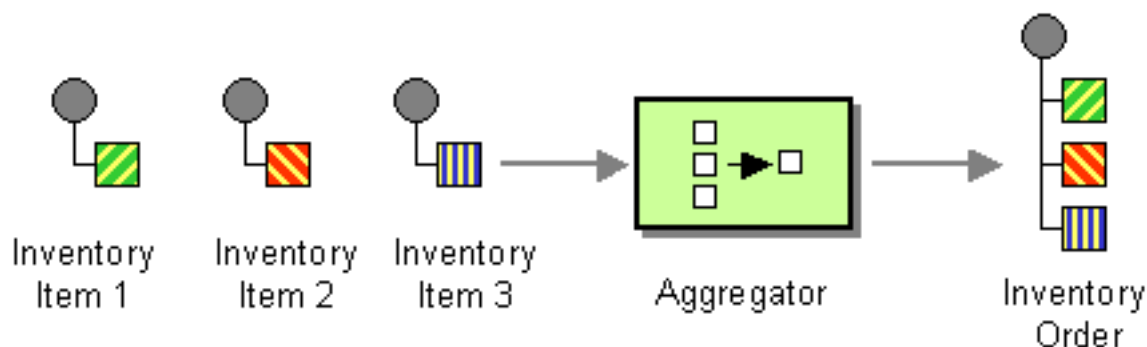
	<i>Selective Consumer</i>	How can a message consumer select which messages it wishes to receive?
	<i>Durable Subscriber</i>	How can a subscriber avoid missing messages while it's not listening for them?
	<i>Idempotent Consumer</i>	How can a message receiver deal with duplicate messages?
	<i>Transactional Client</i>	How can a client control its transactions with the messaging system?
	<i>Messaging Gateway</i>	How do you encapsulate access to the messaging system from the rest of the application?
	<i>Service Activator</i>	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

	<i>Control Bus</i>	How can we effectively administer a messaging system that is distributed across multiple platforms and a wide geographic area?
	<i>Detour</i>	How can you route a message through intermediate steps to perform validation, testing or debugging functions?
	<i>Wire Tap</i>	How do you inspect messages that travel on a point-to-point channel?
	<i>Message History</i>	How can we effectively analyze and debug the flow of messages in a loosely coupled system?
	<i>Log</i>	How can I log processing a message?

2.2. Aggregator

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](#), [LevelDB](#), or [SQL Component](#).

You can manually trigger completion of all current aggregated exchanges by sending 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. Alternatively, starting with Camel 2.11, the header `Exchange.AGGREGATION_COMPLETE_ALL_GROUPS_INCLUSIVE` can be set to `>true` to trigger completion of all groups after processing the current message.

The [Apache Camel website](#) maintains several examples of this EIP in use.

2.2.1. Aggregator options

The aggregator supports the following options:

Option	Default	Description
<code>correlationExpression</code>		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 <code>Exception</code> is thrown. You can disable this by using the <code>ignoreBadCorrelationKeys</code> option.
<code>aggregationStrategy</code>		Mandatory <code>AggregationStrategy</code> which is used to <i>merge</i> the incoming Exchange with the existing already merged exchanges. At first call the <code>oldExchange</code> parameter is <code>null</code> . On subsequent invocations the <code>oldExchange</code> contains the merged exchanges and <code>newExchange</code> is of course the new incoming <code>Exchange</code> . The strategy can also be a <code>TimeoutAwareAggregationStrategy</code> implementation, supporting the <code>timeout</code> callback. Here, Camel will invoke the <code>timeout</code> method when the timeout occurs. Notice that the values for <code>index</code> and <code>total</code> parameters will be <code>-1</code> , and the <code>timeout</code> parameter will only be provided if configured as a fixed value.
<code>strategyRef</code>		A reference to lookup the <code>AggregationStrategy</code> in the Registry . From Camel 2.12 onwards you can also use a POJO as the <code>AggregationStrategy</code> , see further below for details.
<code>strategyMethodName</code>		Camel 2.12: This option can be used to explicit declare the method name to use, when using POJOs as the <code>AggregationStrategy</code> . See further below for more details.
<code>strategyMethodAllowNull</code>	<code>false</code>	Camel 2.12: If this option is <code>false</code> then the <code>aggregate</code> method is not used for the very first aggregation. If this option is <code>true</code> then <code>null</code> values is used as the <code>oldExchange</code> (at the very first aggregation), when

Option	Default	Description
		using POJOs as the <code>AggregationStrategy</code> . See further below for more details.
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 <code>Integer</code> as result. If both are set, Camel will fallback to use the fixed value if the Expression result was <code>null</code> or <code>0</code> .
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 <code>Long</code> as result. If both are set Camel will fallback to use the fixed value if the Expression result was <code>null</code> or <code>0</code> . You cannot use this option together with <code>completionInterval</code> , 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 <code>completionTimeout</code> , only one of them can be used.
completionPredicate		A Predicate to indicate when an aggregated exchange is complete.
completionFromBatchConsumer	false	This option is if the exchanges are coming from a Batch Consumer . Then when enabled the Aggregator will use the batch size determined by the Batch Consumer in the message header <code>CamelBatchSize</code> . 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 <code>completionPredicate</code> option as the Exchange being passed in changes accordingly. When <code>false</code> the Exchange passed in the Predicate is the <i>aggregated</i> Exchange which means any information you may store on the aggregated Exchange from the <code>AggregationStrategy</code> is available for the Predicate . When <code>true</code> the Exchange passed in the Predicate is the <i>incoming</i> 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 <code>org.apache.camel.impl.GroupedExchange</code> 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 <code>AggregationStrategy</code> yourself. Note this option does not support persistent aggregator repositories. See further below for an example and more details.
ignoreInvalidCorrelationKeys	false	Whether or not to ignore correlation keys which could not be evaluated to a value. By default Camel will throw an <code>Exception</code> , but you can enable this option and ignore the situation instead.
closeCorrelationKeyOnCompletion		Whether or not too <i>late</i> 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 <code>closedCorrelationKeyException</code> exception. When using this option you pass in a <code>integer</code> which is a number for a <code>LRUCache</code> which keeps that last X number of closed correlation keys. You can pass in <code>0</code> 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 <code>AggregationRepository</code> class which keeps track of the current inflight aggregated exchanges. Camel uses by default a memory based implementation.

Option	Default	Description
aggregationRepositoryRef		Reference to lookup a aggregationRepository in the Registry .
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.
optimisticLocking	false	Starting with Camel 2.11, turns on using optimistic locking, which requires that the aggregationRepository setting be used.
optimisticLockRetryPolicy		Starting with Camel 2.11.1, allows to configure retry settings when using optimistic locking.

2.2.2. 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, <code>{{forceCompletion}}</code> or interval.

2.2.3. 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.

If your aggregation strategy implements `TimeoutAwareAggregationStrategy`, then 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 be provided only if configured as a fixed value. You must not throw any exceptions from the `timeout` method.

If your aggregation strategy implements `CompletionAwareAggregationStrategy`, then Camel will invoke the `onComplete` method when the aggregated Exchange is completed. This allows you to do any last minute custom logic such as to cleanup some resources, or additional work on the exchange as its now completed. You must not throw any exceptions from the `onCompletion` method.

2.2.4. 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.

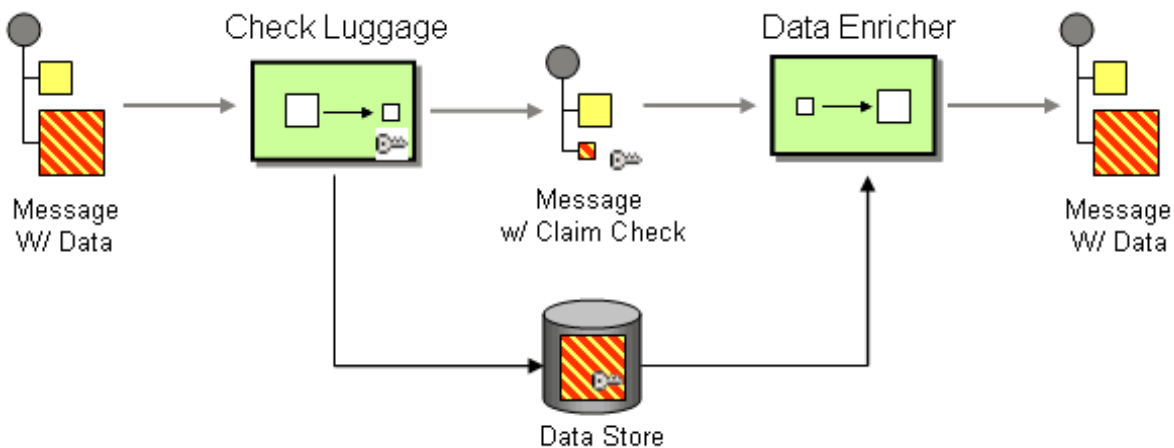
Notice that all the completion ways are per correlation key. And you can combine them in any way you like. It's basically the first which triggers that wins. So you can use a completion size together with a completion timeout. Only `completionTimeout` and `completionInterval` cannot be used at the same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

Notice the completion is a mandatory option and must be provided to the aggregator. If not provided Camel will throw 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.



In the below example we'll 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");
    }
}
```

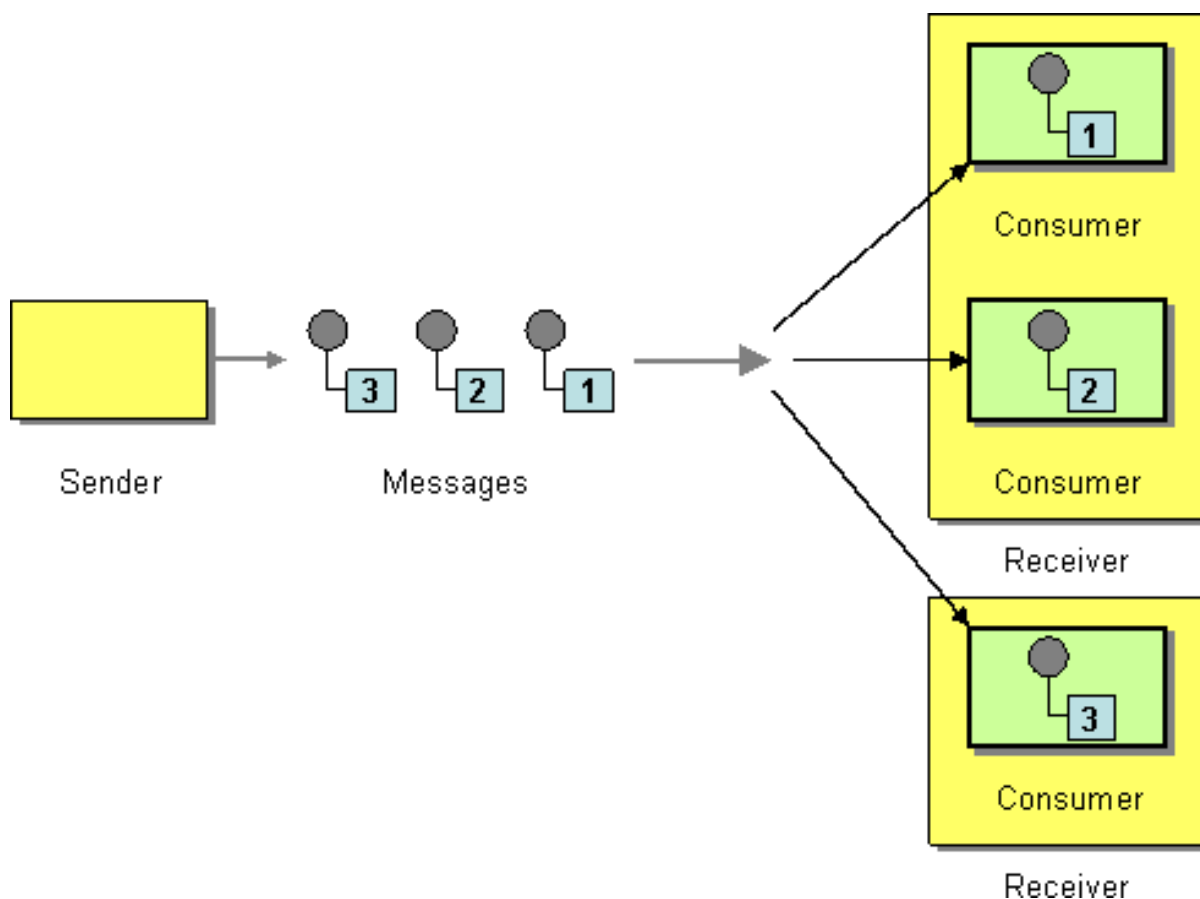
This bean queries the data store using the claim check as the key and then adds the data back into the message. The message body is then removed from the data store and finally the claim check is removed. Now the message is back to what we started with!

For full details, check the example source [here](#):

[camel-core/src/test/java/org/apache/camel/processor/ClaimCheckTest.java](#)

2.4. Competing Consumers

Camel supports the [Competing Consumers](#) from the EIP patterns using a few different components.



You can use the following components to implement competing consumers:-

- [SEDA](#) for SEDA based concurrent processing using a thread pool
- [JMS](#) for distributed SEDA based concurrent processing with queues which support reliable load balancing, failover and clustering.

To enable Competing Consumers with JMS you just need to set the **concurrentConsumers** property on the [JMS](#) endpoint.

For example

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

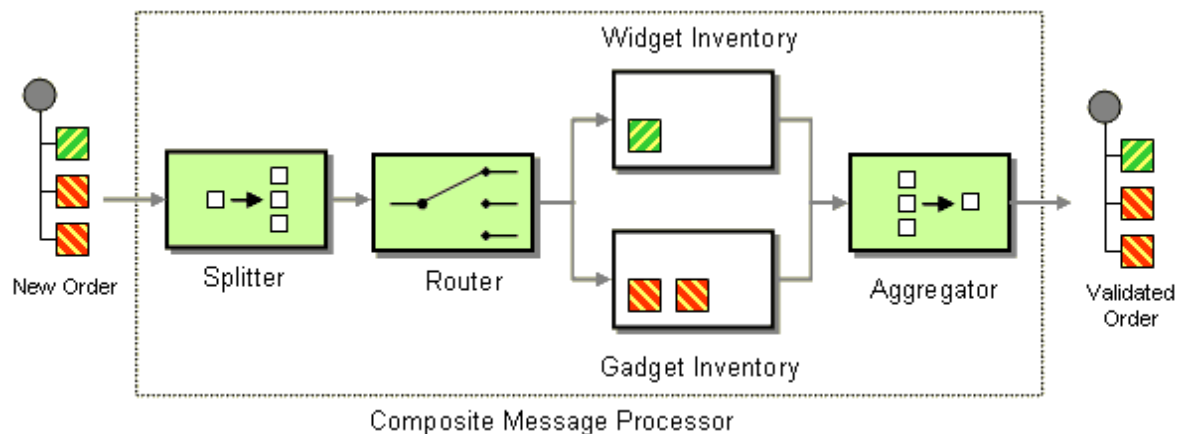
Or in Spring DSL:

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

Or just run multiple JVMs of any [ActiveMQ](#) or [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.

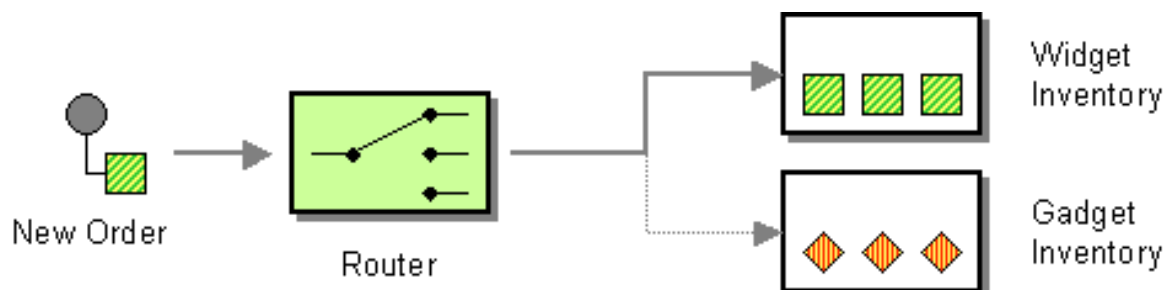


Camel provides two solutions for implementing this EIP -- using both the Splitter and Aggregator EIPs or just the Splitter alone. With the Splitter-only option, all split messages are aggregated back into the same aggregation group (like a fork/join pattern), whereas using an Aggregator provides more flexibility by allowing for grouping into multiple groups. For simplicity, using the Splitter alone should be evaluated first before considering using the Aggregator with it.

See the [Camel Website](#) for the latest examples of this EIP in use.

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

```

Using the Spring XML Extensions

```

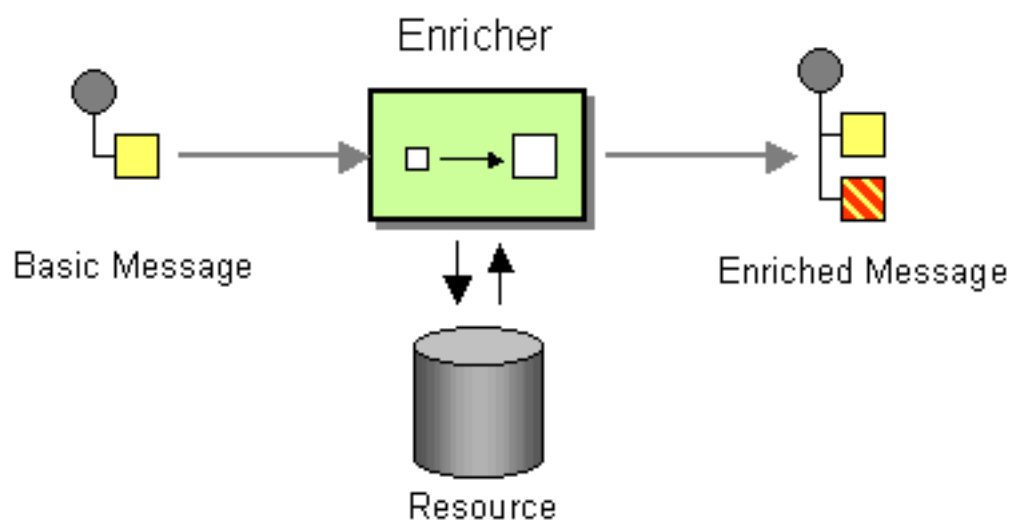
<camelContext errorHandlerRef="errorHandler"
xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:a"/>
        <choice>
            <when>
                <xpath>$foo = 'bar'</xpath>
                <to uri="direct:b"/>
            </when>
            <when>
         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

```

For further examples of this pattern in use see this [JUnit test case](#).

2.7. Content Enricher

Camel supports the [Content Enricher](#) from the EIP patterns using a [Message Translator](#), an arbitrary [Processor](#) in the routing logic, or using the [enrich \[17\]](#) DSL element to enrich the 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 [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 [ActiveMQ](#) 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` uses a `Producer` to obtain the additional data. It is usually used for [Request Reply](#) messaging, for instance to invoke an external web service. `pollEnrich` on the other hand uses a [Polling Consumer](#) to obtain the additional data. It is usually used for [Event Message](#) messaging, for instance to read a file.



*`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.*

Instead of using `enrich` you can use [Recipient List](#) and have dynamic endpoints and define an `AggregationStrategy` on the [Recipient List](#) which then would work as a `enrich` would do.

2.7.3. 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. From Camel 2.12 onwards you can also use a POJO as the AggregationStrategy, see the Aggregate page for more details.
strategyMethodName		Camel 2.12: This option can be used to explicit declare the method name to use, when using POJOs as the AggregationStrategy. See the Aggregate page for more details.
strategyMethodAllowNull	false	Camel 2.12: If this option is false then the aggregate method is not used if there was no data to enrich. If this option is true then null values is used as the oldExchange (when no data to enrich), when using POJOs as the AggregationStrategy. See the Aggregate page for more details.

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.getIn().getBody();
        // combine original body and resourceResponse
        Object mergeResult = ...
        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="..." />
```

2.7.4. 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>
```

2.7.5. Content enrichment using pollEnrich

The `pollEnrich` works just as the `enrich` option however as it uses a [Polling Consumer](#) we have 3 methods when polling

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

2.7.6. PollEnrich Options

Name	Default Value	Description
uri		The endpoint uri for the external service to enrich from. You must use either <code>uri</code> or <code>ref</code> .

Name	Default Value	Description
ref		Refers to the endpoint for the external service to enrich from. You must use either <code>uri</code> or <code>ref</code> .
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. From Camel 2.12 onwards you can also use a POJO as the <code>AggregationStrategy</code> , see the Aggregate page for more details.
strategyMethodName		Camel 2.12: This option can be used to explicit declare the method name to use, when using POJOs as the <code>AggregationStrategy</code> . See the Aggregate page for more details.
strategyMethodAllowNull	false	Camel 2.12: If this option is <code>false</code> then the aggregate method is not used if there was no data to enrich. If this option is <code>true</code> then <code>null</code> values is used as the <code>oldExchange</code> (when no data to enrich), when using POJOs as the <code>AggregationStrategy</code> . See the Aggregate page for more details.
timeout	-1	Timeout in millis when polling from the external service. See below for important details about the timeout.



By default Camel will use the `receive`. Which may block until there is a message available. It is therefore recommended to always provide a timeout value, to make this clear that we may wait for a message, until the timeout is hit.

If there is no data then the `newExchange` in the aggregation strategy is `null`.

You can pass in a timeout value that determines which method to use:

- if timeout is -1 or other negative number then `receive` is selected (**Important:** the `receive` method may block if there is no message)
- if timeout is 0 then `receiveNoWait` is selected
- otherwise `receive(timeout)` is selected

The timeout values is in millis.



`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>
```

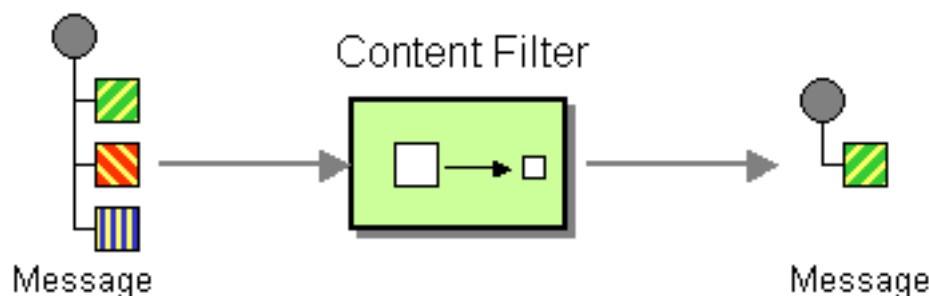
If there is no file then the message is empty. We can use a timeout to either wait (potentially forever) until a file exists, or use a timeout to wait a certain period. For example to wait up to 5 seconds you can do:

```
<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.

- [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

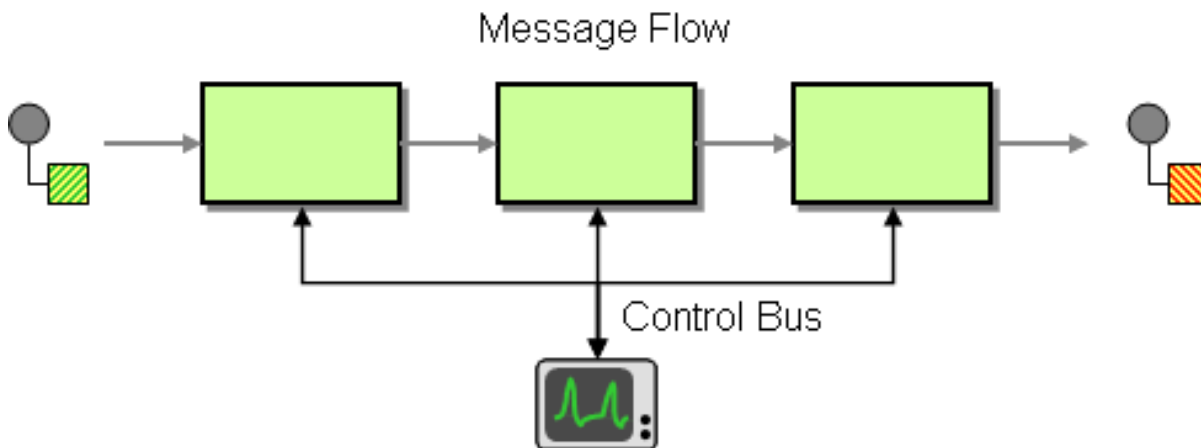
```
<route>
  <from uri="activemq:Input" />
  <bean ref="myBeanName" method="doTransform" />
  <to uri="activemq:Output" />
</route>
```

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. Control Bus

The [Control Bus](#) from the EIP patterns allows for the integration system to be monitored and managed from within the framework.

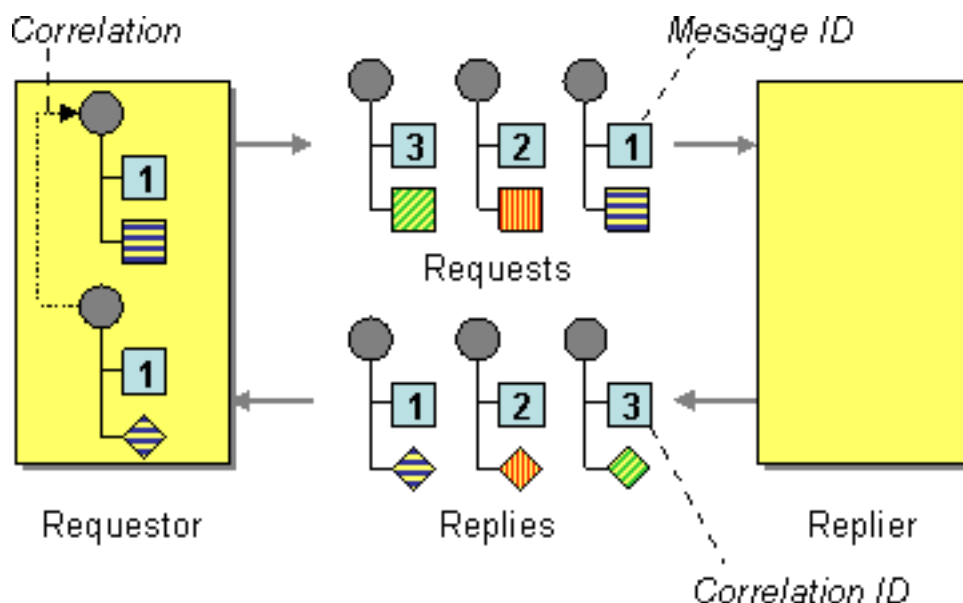


Use a Control Bus to manage an enterprise integration system. The Control Bus uses the same messaging mechanism used by the application data, but uses separate channels to transmit data that is relevant to the management of components involved in the message flow. In Camel you can manage and monitor using JMX, or by using a Java API from the CamelContext, or from the `org.apache.camel.api.management` package, or use the event notifier ([example](#) on the Camel site). Starting with Camel 2.11 a new [ControlBus Component](#) will be available that allows you to send messages to a control bus Endpoint that will react accordingly.

2.10. Correlation Identifier

Camel supports the [Correlation Identifier](#) from the EIP patterns by getting or setting a header on a [Message](#).

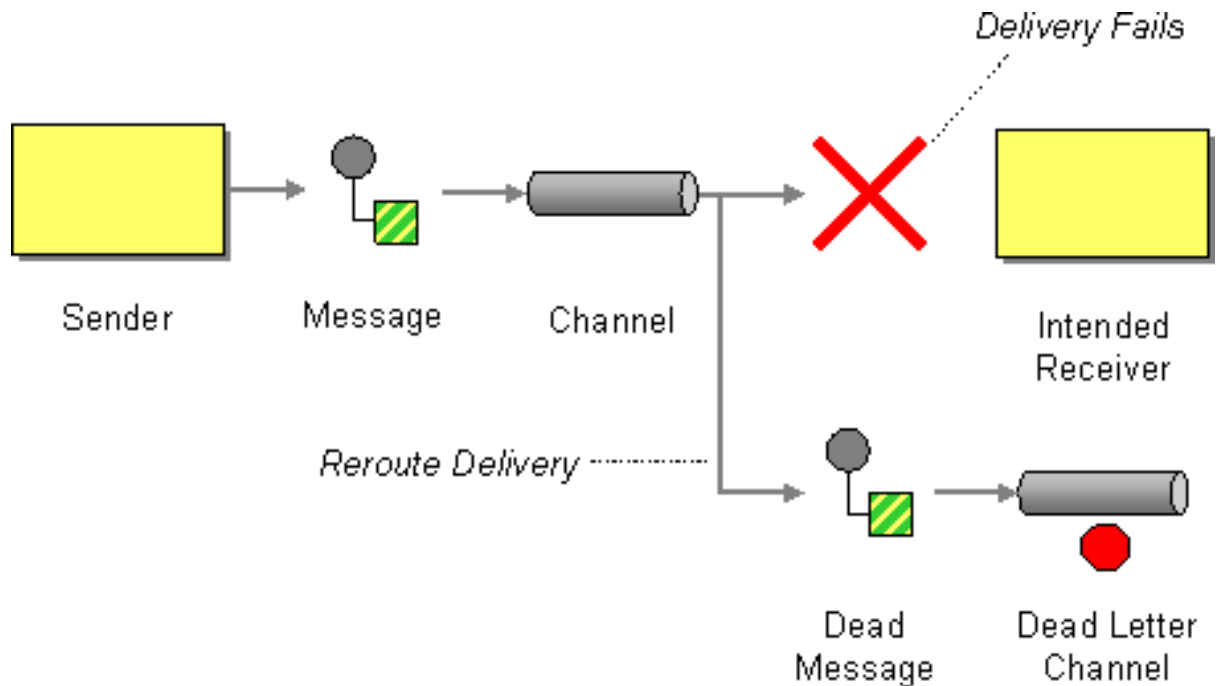
When working with the [ActiveMQ](#) or [JMS](#) components the correlation identifier header is called **JMSCorrelationID**. You can add your own correlation identifier to any message exchange to help correlate messages together to a single conversation (or business process).



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](#)

2.11. Dead Letter Channel

Camel supports the [Dead Letter Channel](#) from the EIP patterns using the `DeadLetterChannel` processor which is an [Error Handler](#). For more information about the `DeadLetterChannel` processor, refer to its corresponding Camel-core API documentation on <http://camel.apache.org/>.



The major difference between [Dead Letter Channel](#) and the [Default Error Handler](#) is that [Dead Letter Channel](#) has a dead letter queue that whenever an [Exchange](#) could not be processed is moved to. It will **always** move 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.

2.11.1. 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.

2.11.2. 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. With this process the Dead Letter Channel has 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 [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.

2.11.3. 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 *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().maximumRedeliveries(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.

2.11.4. OnRedelivery

When *Dead Letter Channel* is doing redelivery 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 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 *Dead Letter Channel* can be viewed as a global scope.

2.11.5. Redelivery default values

Redelivery is disabled by default. The default redelivery 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`.

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.

2.11.6. Redeliver Delay Pattern

Delay pattern is used as a single option to set a range pattern for delays. If used then the following options do 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.

2.11.7. 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.

2.11.8. 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 [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 [Bean](#) processing instead. So be aware 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`.

Starting with Camel 2.11, the route that failed can also be determined by using the following:

```
String failedRouteId = exchange.getProperty(Exchange.FAILURE_ROUTE_ID,
    String.class);
```

2.11.9. Which route failed

Available as of Camel 2.10.4/2.11

When Camel error handler handles an error such as [Dead Letter Channel](#) or using [Exception Clause](#) with `handled=true`, then Camel will decorate the [Exchange](#) with the route id where the error occurred.

```
String failedRouteId = exchange.getProperty(Exchange.FAILURE_ROUTE_ID, String.class);
```

The `Exchange.FAILURE_ROUTE_ID` have the constant value `CamelFailureRouteId`.

This allows for example you to fetch this information in your dead letter queue and use that for error reporting.

2.11.10. 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");
    }
};
```

2.12. 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
<code>asyncDelayed</code>	false	If enabled then delayed messages happens asynchronously using a scheduled thread pool.
<code>executorServiceRef</code>		Refers to a custom Thread Pool to be used if <code>asyncDelay</code> has been enabled.
<code>callerRunsWhenRejected</code>	true	Is used if <code>asyncDelayed</code> was enabled. This controls if the caller thread should execute the task if the thread pool rejected the task.

Using the [Fluent Builders](#)

The example below will delay all messages received on `seda:b` 1 second before sending them to `mock:result`.


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

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 assume 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 [Resequencer](#) with this pattern. For example:

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

You can of course use many different [Expression](#) languages such as [XPath](#), [XQuery](#), [SQL](#) or various [Scripting Languages](#). For example to delay the message for the time period specified in the header, use the following syntax:

```
from("activemq:someQueue").delay(header("delayValue")).to("activemq:aDelayedQueue");
```

And to delay processing using the [Simple](#) language you can use the following DSL:

```
from("activemq:someQueue").delay(simple("${body.delayProperty}")).to("activemq:aDelayedQueue");
```

2.12.1. Spring DSL

The sample below demonstrates the delay in Spring DSL:

```
<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>
```

2.12.2. Asynchronous delaying

You can let the [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.

2.12.2.1. From Java DSL

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

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

```
to("activemq:aDelayedQueue");
```

2.12.2.2. From Spring XML

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

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

2.12.3. Creating a custom delay

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

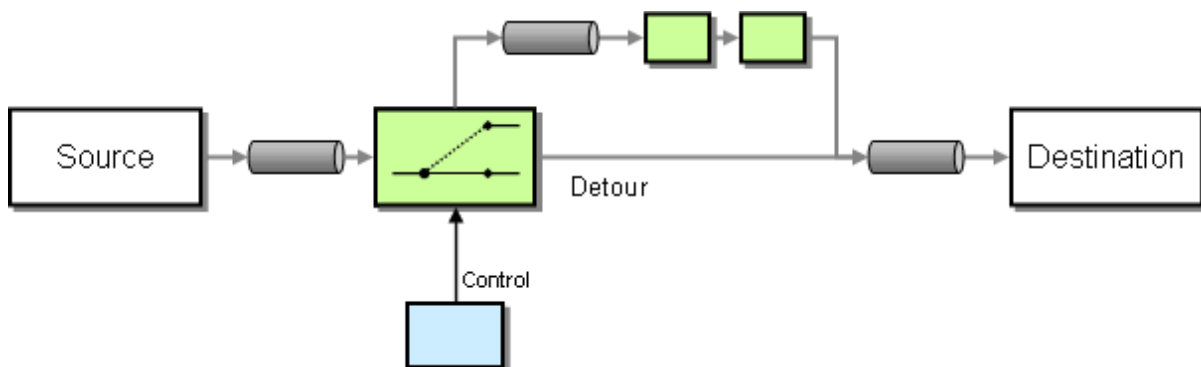
```
from("activemq:foo").
  delay().method("someBean", "computeDelay").
  to("activemq:bar");
```

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;
  }
}
```

2.13. 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.



In the below 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"/>
</route>
```

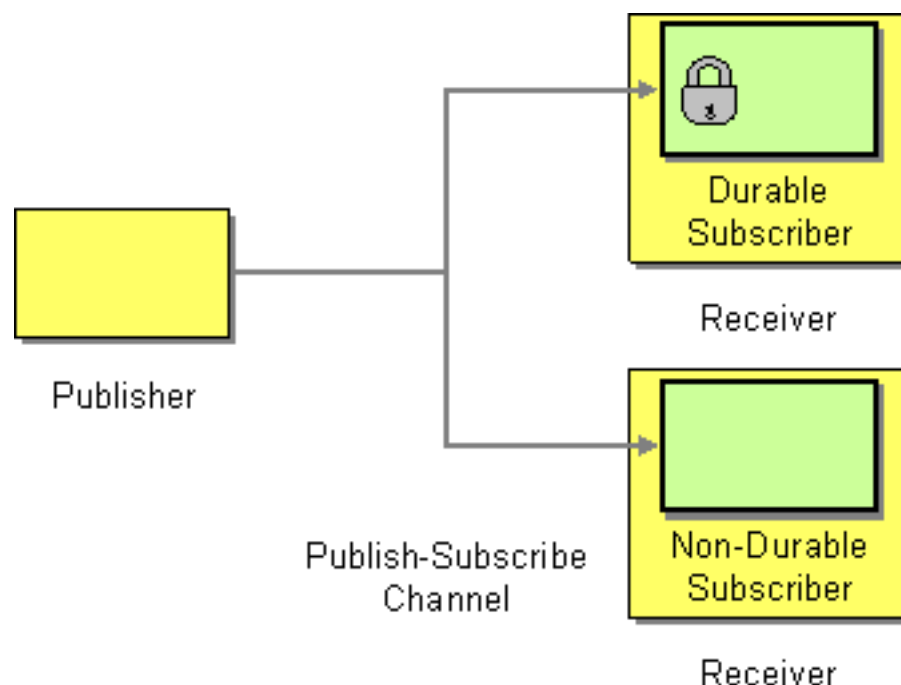
whether the detour is turned on or off is decided by the `ControlBean`. So, when the detour is on the message is routed to `mock:detour` and then `mock:result`. When the detour is off, the message is routed to `mock:result`.

For full details, check the example source here:

[camel-core/src/test/java/org/apache/camel/processor/DetourTest.java](#)

2.14. Durable Subscriber

Camel supports the [Durable Subscriber](#) from the EIP patterns using the [JMS](#) component which supports publish & subscribe using Topics with support for non-durable and durable subscribers.



Another alternative is to combine the [Message Dispatcher](#) or [Content Based Router](#) with [File](#) or [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:

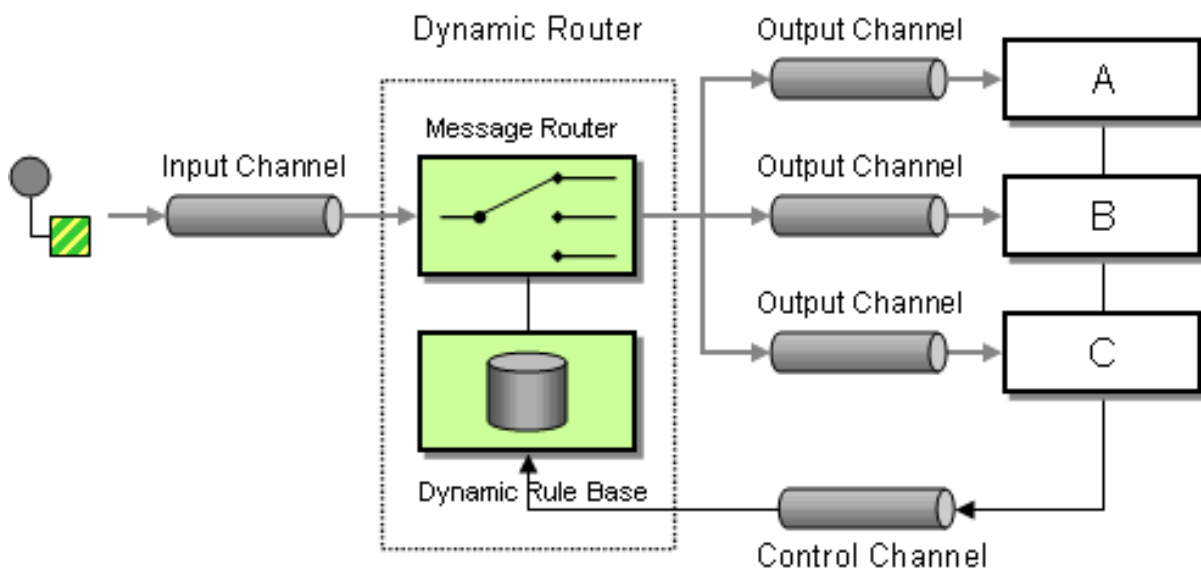
```
<route>
  <from uri="direct:start"/>
  <to uri="activemq:topic:foo"/>
</route>

<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>
```

2.15. 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.



There is a `dynamicRouter` in the DSL which is like a dynamic [Routing Slip](#) which evaluates the slip *on-the-fly*.



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
<code>uriDelimiter</code>	,	Delimiter used if the Expression returned multiple endpoints.
<code>ignoreInvalidEndpoints</code>	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.

Option	Default	Description
cacheSize	1000	Camel 2.13.1/2.12.4: Allows to configure the cache size for the <code>ProducerCache</code> which caches producers for reuse in the routing slip. Will by default use the default cache size which is 1000. Setting the value to -1 allows to turn off the cache all together.

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

2.15.1. 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 [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 null 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.

2.15.2. Spring XML

The same example in Spring XML would be:

```
<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>
```

2.15.3. @DynamicRouter annotation

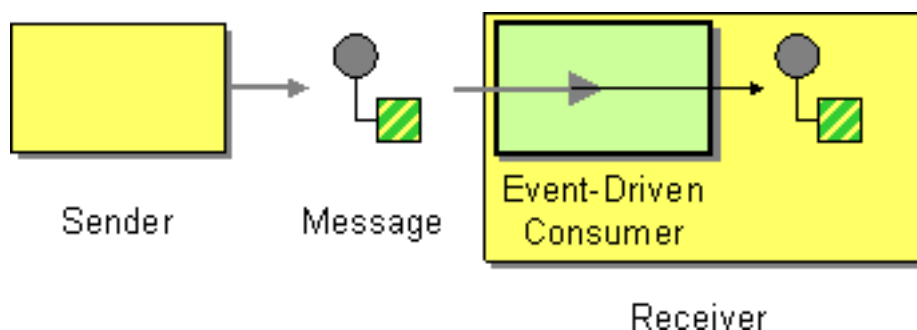
You can also use the `@DynamicRouter` annotation, for example the example below could be written as follows. The `route` method would then be invoked repeatedly as the message is processed dynamically. The idea is to return the next endpoint uri where to go. Return `null` to indicate the end. You can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

```
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.
    }
}
```

2.16. 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 [Message Endpoint](#) when a [Message](#) is available for processing.

For more details see

- [Message](#)

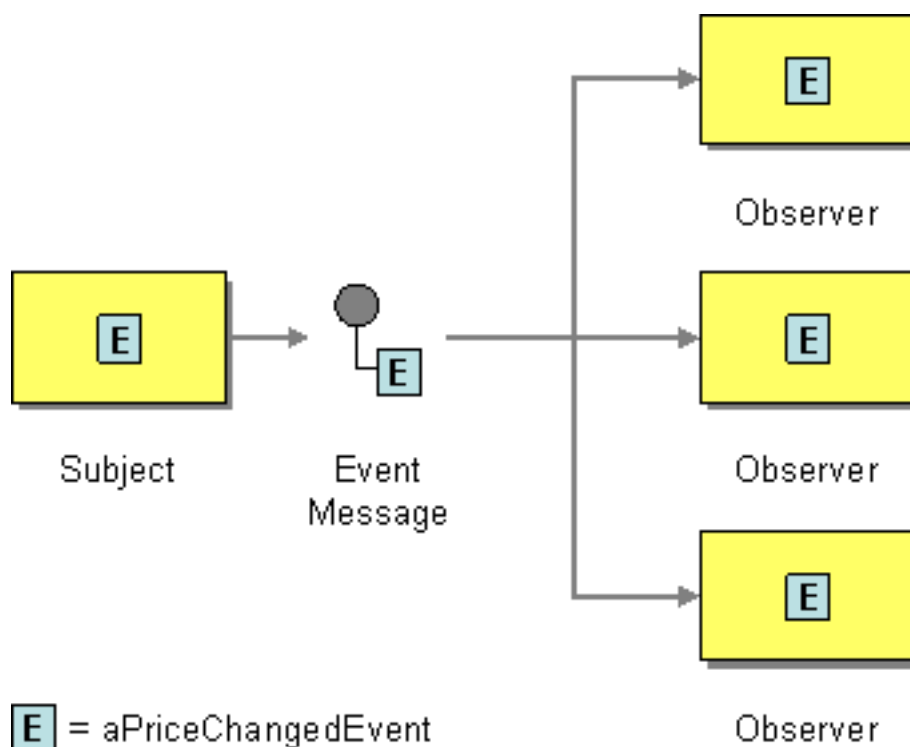
- [Message Endpoint](#)

2.17. Event Message

Camel supports the [Event Message](#) from the EIP patterns by supporting the [Exchange Pattern](#) on a [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.



See also the related [Request Reply](#) EIP.



The default behavior of many Components is InOnly such as for [JMS](#) or [SEDA](#)

If you are using a component which defaults to InOut but wish to use InOnly 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").
    setExchangePattern(ExchangePattern.InOnly).
    bean(Foo.class);
```

or you can invoke an endpoint with an explicit pattern

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

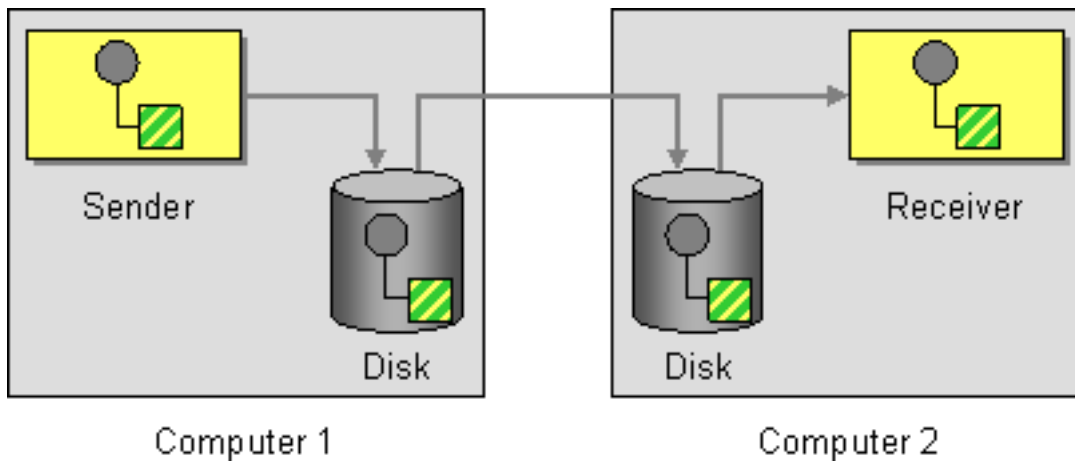
<route>
  <from uri="mq:someQueue"/>
  <inOnly uri="mq:anotherQueue"/>
```

</route>

2.18. Guaranteed Delivery

Camel supports the [Guaranteed Delivery](#) from the EIP patterns using the following components

- [File](#) for using file systems as a persistent store of messages
- [JMS](#) when using persistent delivery (the default) for working with JMS Queues and Topics for high performance, clustering and load balancing
- [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



2.19. 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 [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](#)
- [JpaMessageIdRepository](#)

2.19.1. 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 disabling Camel will only detect duplicates when a message has successfully been processed.
messageIdRepositoryRef	null	A reference to a <code>IdempotentRepository</code> 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.

2.19.2. 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 see this [JUnit test case](#).

2.19.3. 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>

```

2.20. 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.

2.20.1. 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	Camel 2.5: The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to the others. In addition to the weight, endpoint selection is then further refined using round-robin distribution based on weight.
Weighted Random	Camel 2.5: 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	Camel 2.8: From Camel 2.8 onwards the preferred way of using a custom Load Balancer is to use this policy, instead of using the @deprecated <code>ref</code> attribute.
Circuit Breaker	Camel 2.14: Implements the Circuit Breaker pattern as described in "Release it!" book.



If you are proxying and load balancing HTTP, then see [this page](#) for more details.

2.20.2. 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 robin policy. For further examples of this pattern in use see this [JUnit test case](#).

2.20.3. 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`.



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
<code>inheritErrorHandler</code>	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.
<code>maximumFailover-Attempts</code>	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.
<code>roundRobin</code>	boolean	false	Whether or not the <code>failover</code> 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 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 <i>stick</i> 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");
```

2.20.3.1. 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>
```

2.20.3.2. 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>
```

2.20.4. 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.

2.20.5. Circuit Breaker

The Circuit Breaker load balancer is a stateful pattern that monitors all calls for certain exceptions. Initially the Circuit Breaker is in closed state and passes all messages. If there are failures and the threshold is reached, it moves to open state and rejects all calls until halfOpenAfter timeout is reached. After this timeout is reached, if there is a new call, it will pass and if the result is success the Circuit Breaker will move to closed state, or to open state if there was an error.

An example using Java DSL:

```
from("direct:start").loadBalance()
    .circuitBreaker(2, 1000L, MyCustomException.class)
    .to("mock:result");
```

And the same example using Spring XML:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:start"/>
    <loadBalance>
      <circuitBreaker threshold="2" halfOpenAfter="1000">
        <exception>MyCustomException</exception>
      </circuitBreaker>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

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

2.21. Log

How can I log processing a *Message* ?

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

- You can use the *Log* component which logs the Message content.
- You can use the *Tracer* which trace logs message flow.
- You can also use a *Processor* or *Bean* and log from Java code.
- You can use the log DSL, covered below.

The log DSL 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 be 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.



What is the 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 *Log* component is a full fledged component which involves using endpoints and etc. The *Log* component is meant for logging the Message itself and you have many URI options to control what you would like to be logged.



As of **Camel 2.12.4/2.13.1**, if no logger name or logger instance is passed to log DSL, there's a Registry lookup performed to find single instance of `org.slf4j.Logger`. If such instance is found, it is used instead of creating a new logger instance. If more instances are found, the behavior defaults to creating a new instance of logger.



If the message body is stream based, then logging the message body, may cause the message body to be *empty* afterwards. See this [FAQ](#). For streamed messages you can use *Stream caching* to allow logging the message body and be able to read the message body afterwards again.

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");
```

and to set a logger name

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

Since **Camel 2.12.4/2.13.1** the logger instance may be used as well:

```
from("direct:start").log(LoggingLevel.DEBUG,
org.slf4j.LoggerFactory.getLogger("com.mycompany.mylogger"), "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");
```

2.21.1. 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="com.mycompany.MyCoolRoute"/>
  <to uri="mock:baz"/>
</route>
```

Since **Camel 2.12.4/2.13.1** it is possible to reference logger instance. For example:

```
<bean id="myLogger" class="org.slf4j.LoggerFactory"
  factory-method="getLogger" xmlns="http://www.springframework.org/schema/beans">
  <constructor-arg value="com.mycompany.mylogger" />
</bean>

<route id="moo" xmlns="http://camel.apache.org/schema/spring">
  <from uri="direct:moo"/>
  <log message="Me Got ${body}" loggingLevel="INFO" loggerRef="myLogger"/>
  <to uri="mock:baz"/>
</route>
```

2.21.2. 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="com.mycompany.MyCoolRoute"
    marker="myMarker"/>
  <to uri="mock:baz"/>
</route>
```

2.21.3. Using log DSL in OSGi

Improvement as of Camel 2.12.4/2.13.1

When using log DSL inside OSGi (e.g., in Karaf), the underlying logging mechanisms are provided by PAX logging. It searches for a bundle which invokes `org.slf4j.LoggerFactory.getLogger()` method and associates the bundle with the logger instance. Passing only logger name to log DSL results in associating `camel-core` bundle with the logger instance created.

In some scenarios it is required that the bundle associated with logger should be the bundle which contains route definition. This is possible using provided logger instance both for Java DSL and Spring DSL (see the examples above).

2.22. 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 *Message* in different ways.

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

that could be used by processors down the pipeline to process the *Message* in different ways.

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 `RuntimeException` 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>
```

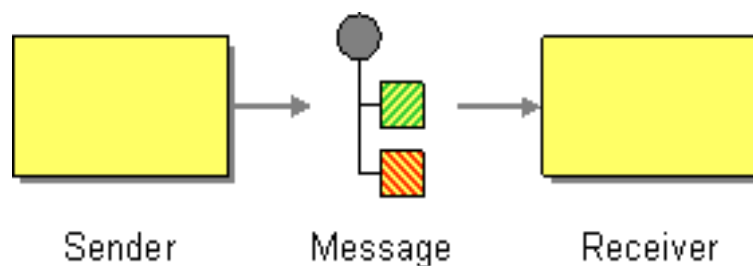


```
</loop>
</route>
```

See the [Camel Website](#) for further examples of this pattern in use.

2.23. Message

Camel supports the [Message](#) from the EIP patterns using the [Message](#) interface.



To support various message exchange patterns like one way [Event Message](#) and [Request Reply](#) messages Camel uses an [Exchange](#) interface which has a **pattern** property which can be set to **InOnly** for an [Event Message](#) which has a single inbound Message, or **InOut** for a [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");
```

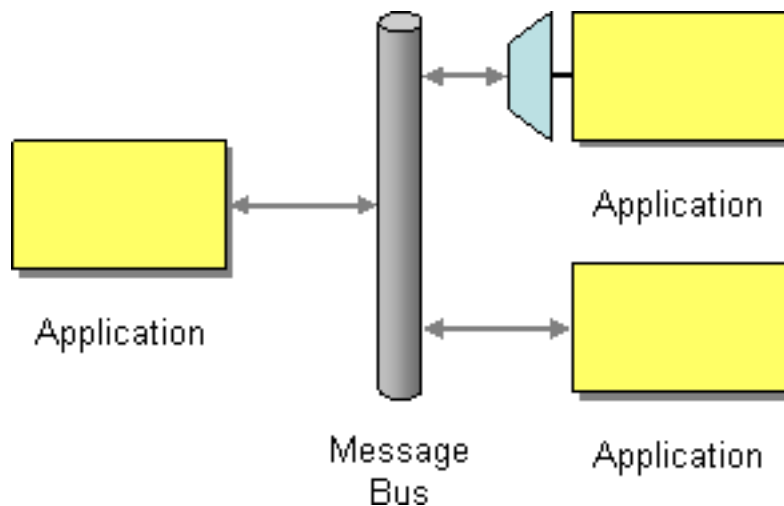
Route Using the Spring XML Extensions

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

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

2.24. 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 [JMS](#) component for traditional MOM support.

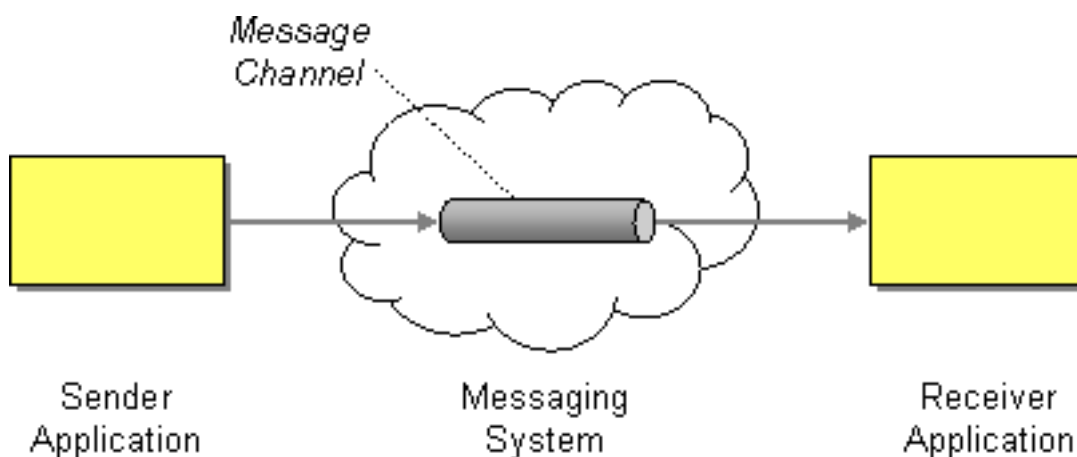
Also worthy of note is the [XMPP](#) component for supporting messaging over XMPP (Jabber)

Of course there are also ESB products such as [Apache ServiceMix](#) which serve as full fledged message busses.

You can interact with [Apache ServiceMix](#) from Camel in many ways, but in particular you can use the [NMR](#) or [JBI](#) component to access the ServiceMix message bus directly.

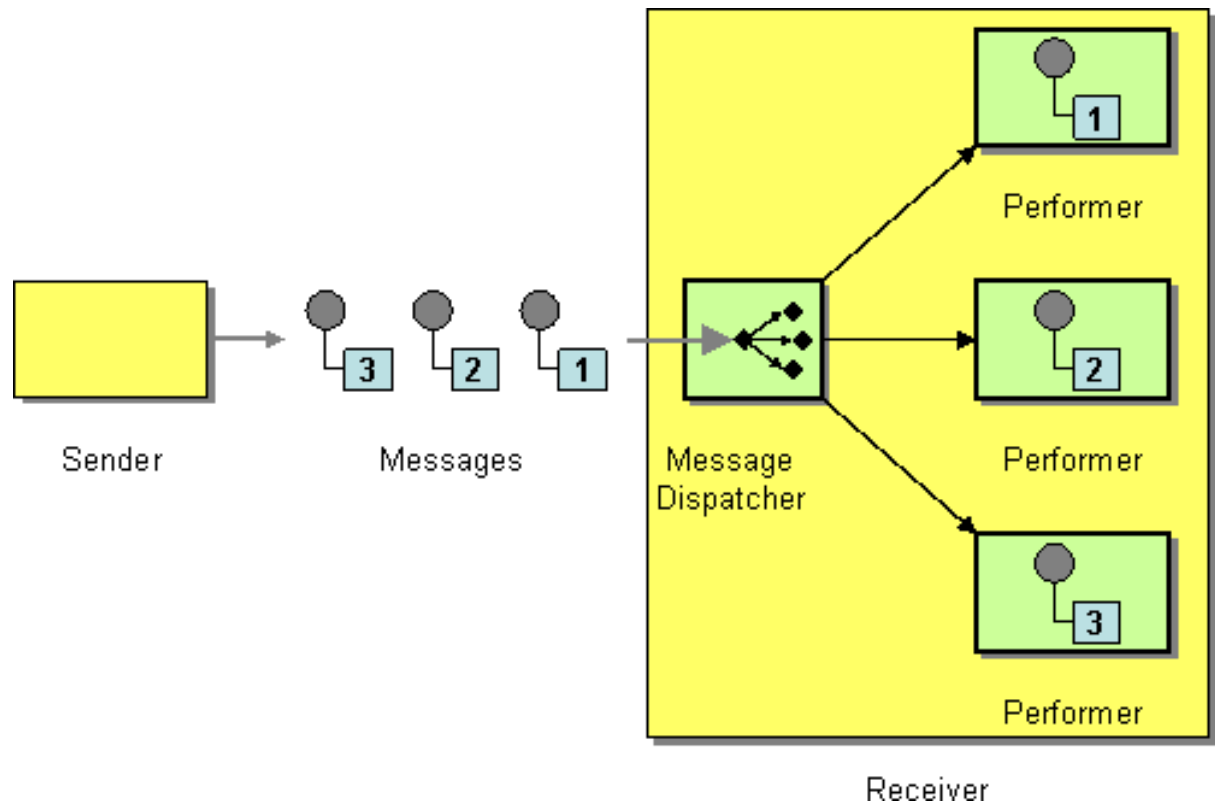
2.25. Message Channel

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 [Message](#) and [Message Endpoint](#).



2.26. Message Dispatcher

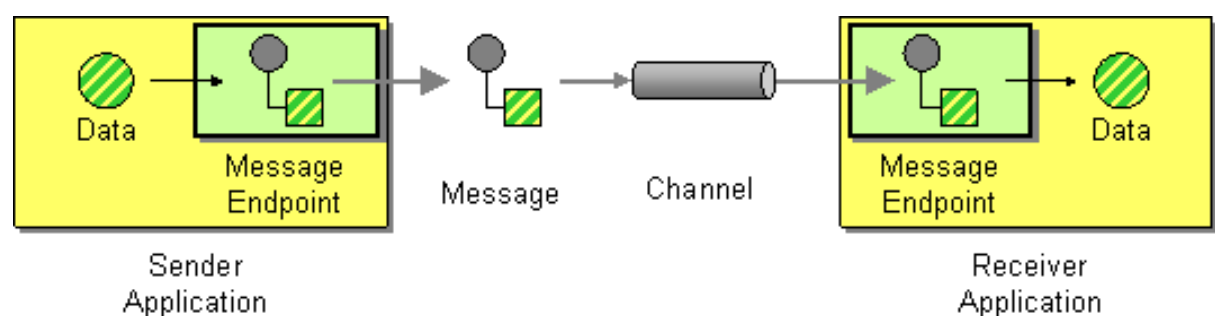
Camel supports the [Message Dispatcher](#) from the EIP patterns using various approaches.



You can use a component like *JMS* with selectors to implement a *Selective Consumer* as the Message Dispatcher implementation. Or you can use an *Endpoint* as the Message Dispatcher itself and then use a *Content Based Router* as the Message Dispatcher.

2.27. Message Endpoint

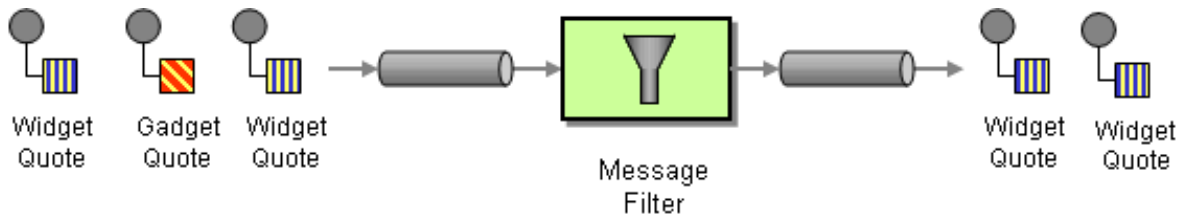
Camel supports the *Message Endpoint* from the EIP patterns using the *Endpoint* interface.



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.

2.28. 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>
```

You can also use a method call expression (to call a method on a bean) in the Message Filter, as shown below:

```
<bean id="myBean" class="com.foo.MyBean"/>
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:a"/>
        <filter>
            <method ref="myBean" method="isGoldCustomer"/>
            <to uri="direct:b"/>
        </filter>
    </route>
</camelContext>
```

For further examples of this pattern in use see this [JUnit test case](#).

2.28.1. 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 [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");
```

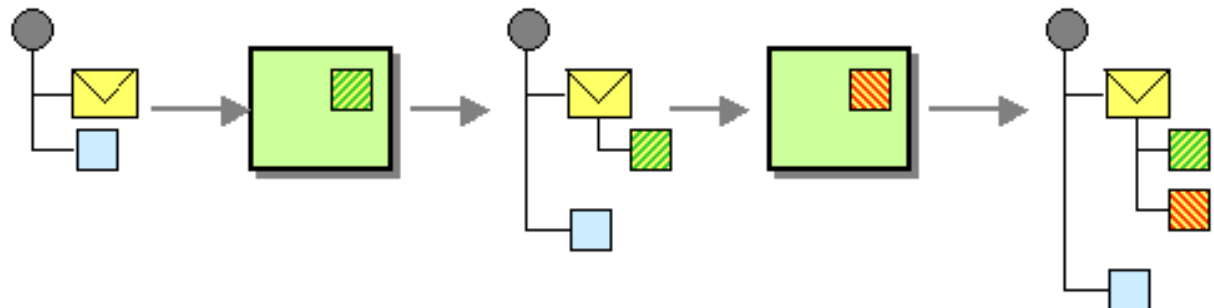
2.28.2. Knowing if Exchange was filtered or not

The Message Filter EIP will add a property on the [Exchange](#) that states if it was filtered or not.

The property has the key `Exchange.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. This property will be visible within the [Message Filter](#) block who's [Predicate](#) matches (value set to `true`), and to the steps immediately following the [Message Filter](#) with the value set based on the results of the last [Message FilterPredicate](#) evaluated.

2.29. Message History

The [Message History](#) from the EIP patterns allows for analyzing and debugging the flow of messages in a loosely coupled system.



Attaching a Message History to the message will provide a list of all applications that the message passed through since its origination. In Camel you can trace message flow using the [Tracer](#), or access information using the Java API from [UnitOfWork](#) using the `getTracedRouteNodes` method. When Camel sends a message to an endpoint that endpoint information is stored on the [Exchange](#) as a property with the key `Exchange.TO_ENDPOINT`. This property contains the last known endpoint the [Exchange](#) was sent to (it will be overridden when sending to new endpoint). Alternatively you can trace messages being sent using [interceptors](#) or the [Event Notifier](#).

2.29.1. Easier Message History

Available as of Camel 2.12

[Message History](#) is enabled by default from Camel 2.12. During routing Camel captures how the [Exchange](#) is routed, as a `org.apache.camel.MessageHistory` entity that is stored on the [Exchange](#). On the `org.apache.camel.MessageHistory` there is information about the route id, processor id, timestamp, and elapsed time it took to process the [Exchange](#) by the processor.

The information can be reached from Java code with:

```
List<MessageHistory> list = exchange.getProperty(Exchange.MESSAGE_HISTORY, List.class);
...
```

2.29.1.1. Enabling or disabling message history

The [Message History](#) can be enabled or disabled per `CamelContext` or per route. For example you can turn it off with

```
camelContext.setMessageHistory(false);
```

Or from XML DSL with

```
<camelContext messageHistory="false" ...>
...
</camelContext>
```

You can also do this per route. Then a route level configuration overrides the `CamelContext` level configuration.

2.29.1.2. Route stack-trace in exceptions logged by error handler

If [Message History](#) is enabled, then Camel will leverage this information, when the [Error Handler](#) logs exhausted exceptions. Then in addition to the caused exception with its stacktrace, you can see the message history; you may think this as a "route stacktrace". And example is provided below:

```
2013-05-31 14:41:28,084 [ - seda://start] ERROR
DefaultErrorHandler - Failed delivery for (MessageId:
ID-davsclaus-air-lan-55446-1370004087263-0-1 on ExchangeId:
ID-davsclaus-air-lan-55446-1370004087263-0-3). Exhausted after delivery
attempt: 1 caught: java.lang.IllegalArgumentException: Forced to dump
message history

Message History
-----
RouteId ProcessorId Processor
Elapsed (ms)
[route1] [to1      ] [log:foo
] [ 6]
[route1] [to2      ] [direct:bar
] [102]
[route2] [to5      ] [log:bar
] [ 1]
[route2] [delay2    ] [delay[{100}]
] [100]
[route2] [to6      ] [mock:bar
] [ 0]
[route1] [delay1    ] [delay[{300}]
] [303]
[route1] [to3      ] [log:baz
] [ 0]
[route1] [process1] [org.apache.camel.processor.MessageHistoryDumpRoutingTest
$l$l@6a53f9d8] [ 2]

Stacktrace
-----
java.lang.IllegalArgumentException: Forced to dump message history
    at org.apache.camel.processor.MessageHistoryDumpRoutingTest$l$l.process
(MessageHistoryDumpRoutingTest.java:54)
    at org.apache.camel.processor.DelegateSyncProcessor.process
(DelegateSyncProcessor.java:63)
```

```

at org.apache.camel.processor.RedeliveryErrorHandler.process
(RedeliveryErrorHandler.java:388)
at org.apache.camel.processor.CamelInternalProcessor.process
(CamelInternalProcessor.java:189)
at org.apache.camel.processor.Pipeline.process(Pipeline.java:118)
at org.apache.camel.processor.Pipeline.process(Pipeline.java:80)
at org.apache.camel.processor.DelayProcessorSupport.process
(DelayProcessorSupport.java:117)
at org.apache.camel.processor.RedeliveryErrorHandler.process
(RedeliveryErrorHandler.java:388)
at org.apache.camel.processor.CamelInternalProcessor.process
(CamelInternalProcessor.java:189)
at org.apache.camel.processor.Pipeline.process(Pipeline.java:118)
at org.apache.camel.processor.Pipeline.process(Pipeline.java:80)
at org.apache.camel.processor.CamelInternalProcessor.process
(CamelInternalProcessor.java:189)
at org.apache.camel.component.seda.SedaConsumer.sendToConsumers
(SedaConsumer.java:293)
at org.apache.camel.component.seda.SedaConsumer.doRun(SedaConsumer.java:202)
at org.apache.camel.component.seda.SedaConsumer.run(SedaConsumer.java:149)
at java.util.concurrent.ThreadPoolExecutor.runWorker
(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run
(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)

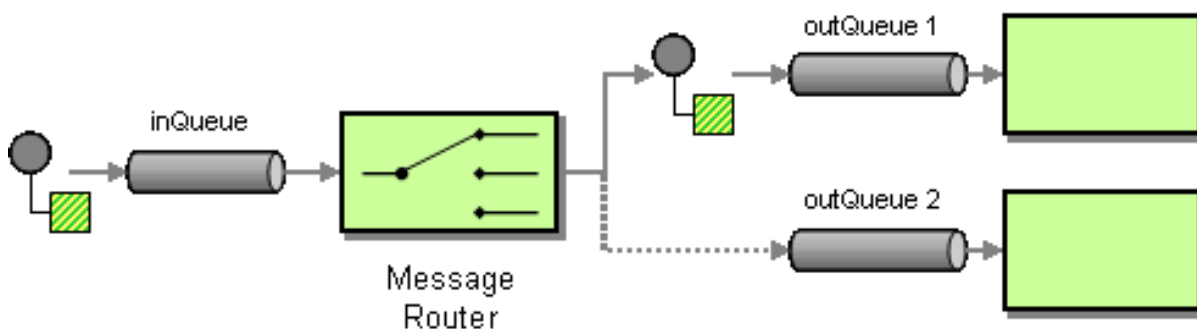
```

You can turn off logging message history from the [Error Handler](#) using

```
errorHandler(defaultErrorHandler().logExhaustedMessageHistory(false));
```

2.30. 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");
    }
}
```

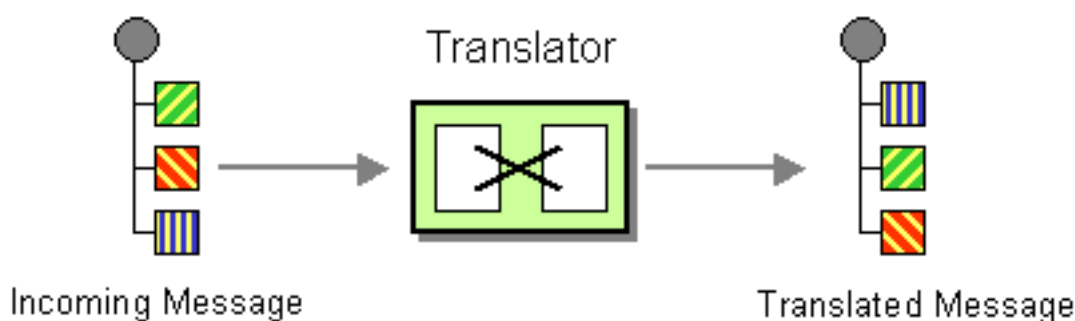
Using the [Spring XML Extensions](#)

```
<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.

2.31. 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.



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", "methodName").
    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 [Velocity](#) or [XQuery](#) and then send it on to another destination. For example using InOnly (one way messaging)

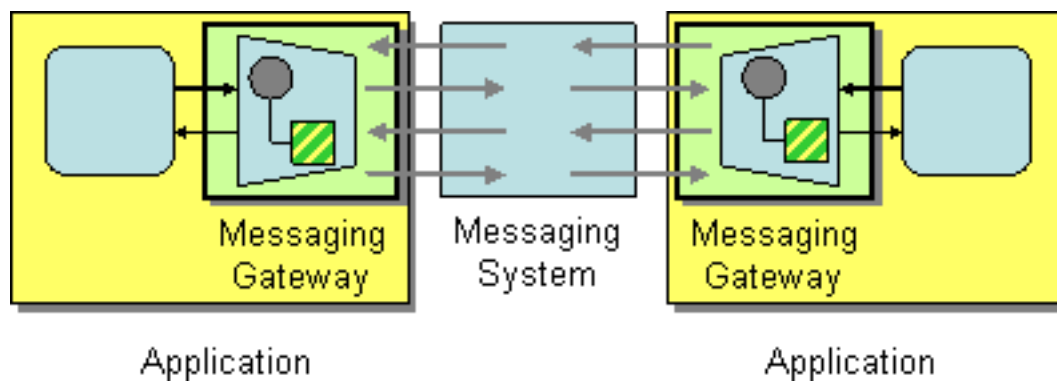
```
from("activemq:My.Queue").
    to("velocity:com/acme/MyResponse.vm").
    to("activemq:Another.Queue");
```

If you want to use InOut (request-reply) semantics to process requests on the **My.Queue** queue on [ActiveMQ](#) 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");
```

2.32. Messaging Gateway

Camel has several endpoint components that support the [Messaging Gateway](#) from the EIP patterns.

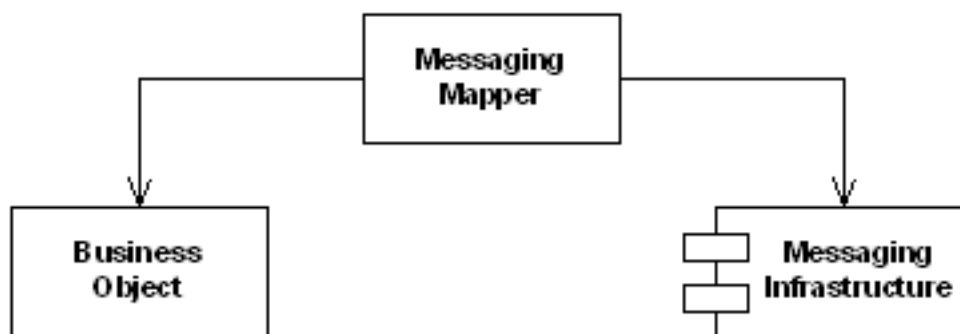


Components like [Bean](#) and [CXF](#) provide a way to bind a Java interface to the message exchange.

However you may want to read the [Using CamelProxy](#) documentation as a true [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 [Messaging Gateway](#) EIP solution.

2.33. Messaging Mapper

Camel supports the [Messaging Mapper](#) from the EIP patterns by using either [Message Translator](#) pattern or the [Type Converter](#) module.



2.34. 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. Options:

Name	Default Value	Description
<code>strategyRef</code>		Refers to an <code>AggregationStrategy</code> 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. From Camel 2.12 onwards you can also use a POJO as the <code>AggregationStrategy</code> , see the Aggregate page for more details.
<code>strategyMethodName</code>		Camel 2.12: This option can be used to explicit declare the method name to use, when using POJOs as the <code>AggregationStrategy</code> . See the Aggregate page for more details.
<code>strategyMethodAllowNull</code>	false	Camel 2.12: If this option is false

Name	Default Value	Description
		then the aggregate method is not used if there was no data to enrich. If this option is <code>true</code> then <code>null</code> values is used as the <code>oldExchange</code> (when no data to enrich), when using POJOs as the <code>AggregationStrategy</code> . See the Aggregate page for more details.
<code>parallelProcessing</code>	<code>false</code>	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.
<code>parallelAggregate</code>	<code>false</code>	Camel 2.14: If enabled then the <code>aggregate</code> method on <code>AggregationStrategy</code> can be called concurrently. Notice that this would require the implementation of <code>AggregationStrategy</code> to be implemented as thread-safe. By default this is <code>false</code> meaning that Camel synchronizes the call to the <code>aggregate</code> method. Though in some use-cases this can be used to archive higher performance when the <code>AggregationStrategy</code> is implemented as thread-safe.
<code>executorServiceRef</code>		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.
<code>stopOnException</code>	<code>false</code>	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 <code>AggregationStrategy</code> class where you have full control how to handle that.
<code>streaming</code>	<code>false</code>	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.
<code>timeout</code>		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 <code>TimeoutAwareAggregationStrategy</code> then the <code>timeout</code> method is invoked before breaking out. If the timeout is reached with running tasks still remaining, certain tasks for which it is difficult for Camel to shut down in a graceful manner may continue to run. So use this option with a bit of care.
<code>onPrepareRef</code>		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.
<code>shareUnitOfWork</code>	<code>false</code>	Whether the unit of work should be shared. See the same option on <code>Splitter</code> for more details.

2.34.1. 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");
```

2.34.2. Stop processing in case of exception

The *Multicast* will by default continue to process the entire *Exchange* even in case one of the multicasted messages will throw 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>
```

2.34.3. Using onPrepare to execute custom logic when preparing messages

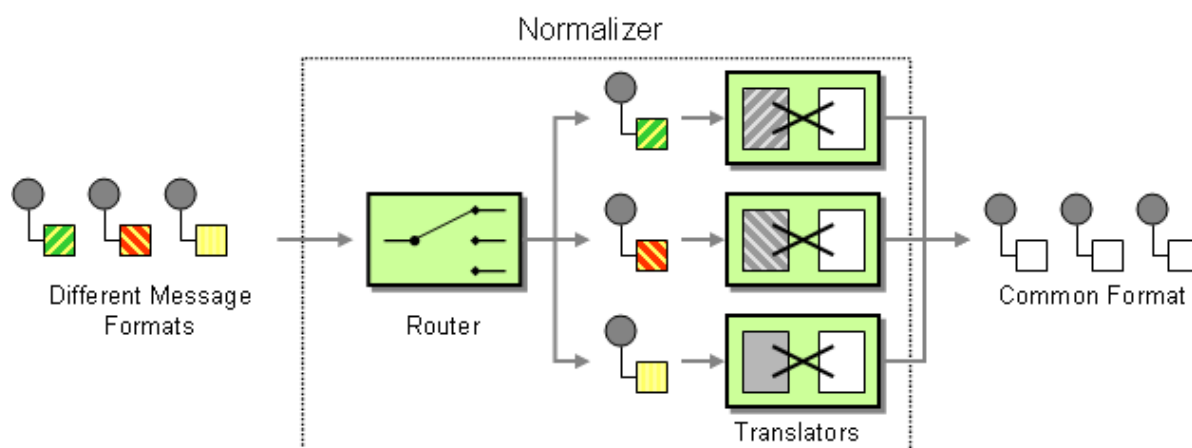
The Multicast will copy the source Exchange and multicast each copy. However the copy is a shallow copy, so in case you have mutable message bodies, then any changes will be visible by the other copied messages. If you want to use a deep clone copy then you need to use a custom `onPrepare` which allows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

2.35. Normalizer

Camel supports the [Normalizer](#) from the EIP patterns by using a [Message Router](#) in front of a number of [Message Translator](#) instances.



The below 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](#)

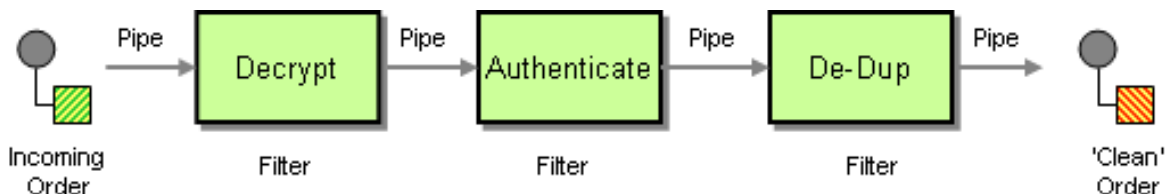
The same example in the Spring DSL

```
<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" />
```

2.36. 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.

You can create pipelines of logic using multiple Endpoint or [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:

```
<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.

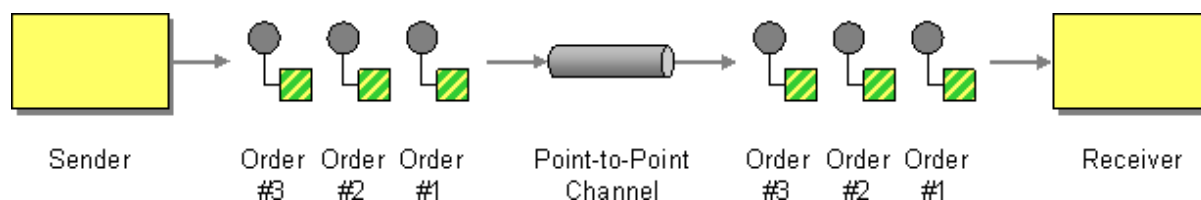
```
<route>
  <from uri="activemq:SomeQueue"/>
  <multicast>
    <pipeline>
      <bean ref="something"/>
      <to uri="log:Something"/>
    </pipeline>
    <pipeline>
      <bean ref="foo"/>
      <bean ref="bar"/>
      <to uri="activemq:OutputQueue"/>
    </pipeline>
  </multicast>
</route>
```

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

2.37. Point to Point Channel

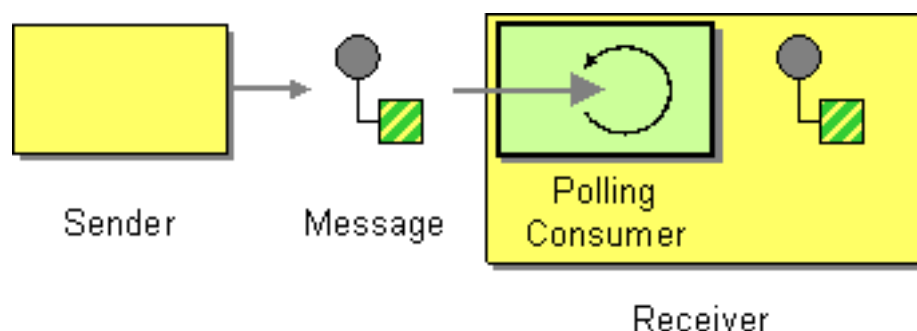
Camel supports the [Point to Point Channel](#) from the EIP patterns using the following components

- [SEDA](#) for in-VM seda based messaging
- [JMS](#) for working with JMS Queues for high performance, clustering and load balancing
- [JPA](#) for using a database as a simple message queue
- [XMPP](#) for point-to-point communication over XMPP (Jabber)
- and others



2.38. 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();
```

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

2.38.1. EventDrivenPollingConsumer Options

The EventDrivePollingConsumer (the default implementation) supports the following options:

Option	Default	Description
pollingConsumerQueueSize	1000	Camel 2.14/2.13.1/2.12.4: The queue size for the internal handoff queue between the polling consumer, and producers sending data into the queue.
pollingConsumerBlockWhenFull	True	Camel 2.14/2.13.1/2.12/4: Whether to block any producer if the internal queue is full.

Notice that some Camel [Components](#) has their own implementation of `PollingConsumer` and therefore do not support the options above.

You can configure these options in endpoints [URIs](#), such as shown below:

```
Endpoint endpoint = context.getEndpoint("file:inbox?pollingConsumerQueueSize=50");
PollingConsumer consumer = endpoint.createPollingConsumer();
Exchange exchange = consumer.receive(5000);
```

2.38.2. ConsumerTemplate

The `ConsumerTemplate` is a template much like Spring's `JmsTemplate` or `JdbcTemplate` supporting the [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.

2.38.3. 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 [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 <code>org.apache.camel.PollingConsumerPollingStrategy</code> 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 <code>WARN</code> level and ignore it.
sendEmptyMessage-WhenIdle	false	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.
initialDelay	1000	Milliseconds before the first poll starts.
delay	500	Milliseconds before the next poll of the file/directory.
useFixedDelay	true	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.
timeUnit	<code>TimeUnit.MILLISECONDS</code>	Time unit for <code>initialDelay</code> and <code>delay</code> options.
runLoggingLevel	TRACE	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.
scheduledExecutor-Service	null	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool. This option allows you to share a thread pool among multiple consumers.
greedy	false	Camel 2.10.6/2.11.1: If greedy is enabled, then the <code>ScheduledPollConsumer</code> will run immediately again, if the previous run polled 1 or more messages.
scheduler	null	Camel 2.12: Allow to plugin a custom <code>org.apache.camel.spi.ScheduledPollConsumerScheduler</code> to use as the scheduler for firing when the polling consumer runs. The default implementation uses the <code>ScheduledExecutorService</code> and there is a Quartz2 , and Spring based which supports CRON expressions. Notice: If using a custom scheduler then the options for <code>initialDelay</code> , <code>useFixedDelay</code> , <code>timeUnit</code> , and <code>scheduledExecutorService</code> may not be in use. Use the text <code>quartz2</code> to refer to use the Quartz2 scheduler; and use the text <code>spring</code> to use the Spring based; and use the text <code>#myScheduler</code> to refer to a custom scheduler by its id in the Registry . See Quartz2 page for an example.
scheduler.xxx	null	Camel 2.12: To configure additional properties when using a custom scheduler or any of the Quartz2 , Spring based scheduler.
backoffMultiplier	0	Camel 2.12: To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next

Option	Default	Description
		actual attempt is happening again. When this option is in use then <code>backoffIdleThreshold</code> and/or <code>backoffErrorThreshold</code> must also be configured.
<code>backoffIdleThreshold</code>	0	Camel 2.12: The number of subsequent idle polls that should happen before the <code>backoffMultiplier</code> should kick-in.
<code>backoffErrorThreshold</code>	0	Camel 2.12: The number of subsequent error polls (failed due some error) that should happen before the <code>backoffMultiplier</code> should kick-in.

2.38.4. Using backoff to let the scheduler be less aggressive

Available as of Camel 2.12

The scheduled [Polling Consumer](#) is by default static by using the same poll frequency whether or not there is messages to pickup or not. From Camel 2.12 onwards you can configure the scheduled [Polling Consumer](#) to be more dynamic by using backoff. This allows the scheduler to skip N number of polls when it becomes idle, or there has been X number of errors in a row. See more details in the table above for the `backoffXXX` options.

For example to let a FTP consumer backoff if its becoming idle for a while you can do:

```
from( "ftp://myserver?username=foo&passowrd=secret?
delete=true&delay=5s&backoffMultiplier=6&backoffIdleThreshold=5" )
.to( "bean:processFile" );
```

In this example, the FTP consumer will poll for new FTP files evert 5th second. But if it has been idle for 5 attempts in a row, then it will backoff using a multiplier of 6, which means it will now poll every $5 \times 6 = 30$ th second instead. When the consumer eventually pickup a file, then the backoff will reset, and the consumer will go back and poll every 5th second again.

Camel will log at `DEBUG` level using `org.apache.camel.impl.ScheduledPollConsumer` when backoff is kicking-in.

2.38.5. 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.

2.38.6. Using a custom scheduler

Available as of Camel 2.12:

The SPI interface `org.apache.camel.spi.ScheduledPollConsumerScheduler` allows to implement a custom scheduler to control when the [Polling Consumer](#) runs. The default implementation is based on the JDKs

`ScheduledExecutorService` with a single thread in the thread pool. There is a CRON based implementation in the [Quartz2](#), and [Spring](#) components.

For an example of developing and using a custom scheduler, see the unit test `org.apache.camel.component.file.FileConsumerCustomSchedulerTest` from the source code in `camel-core`.

2.38.6.1. 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.

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.

2.38.6.2. Configuring an Endpoint to use `PollingConsumerPollStrategy`

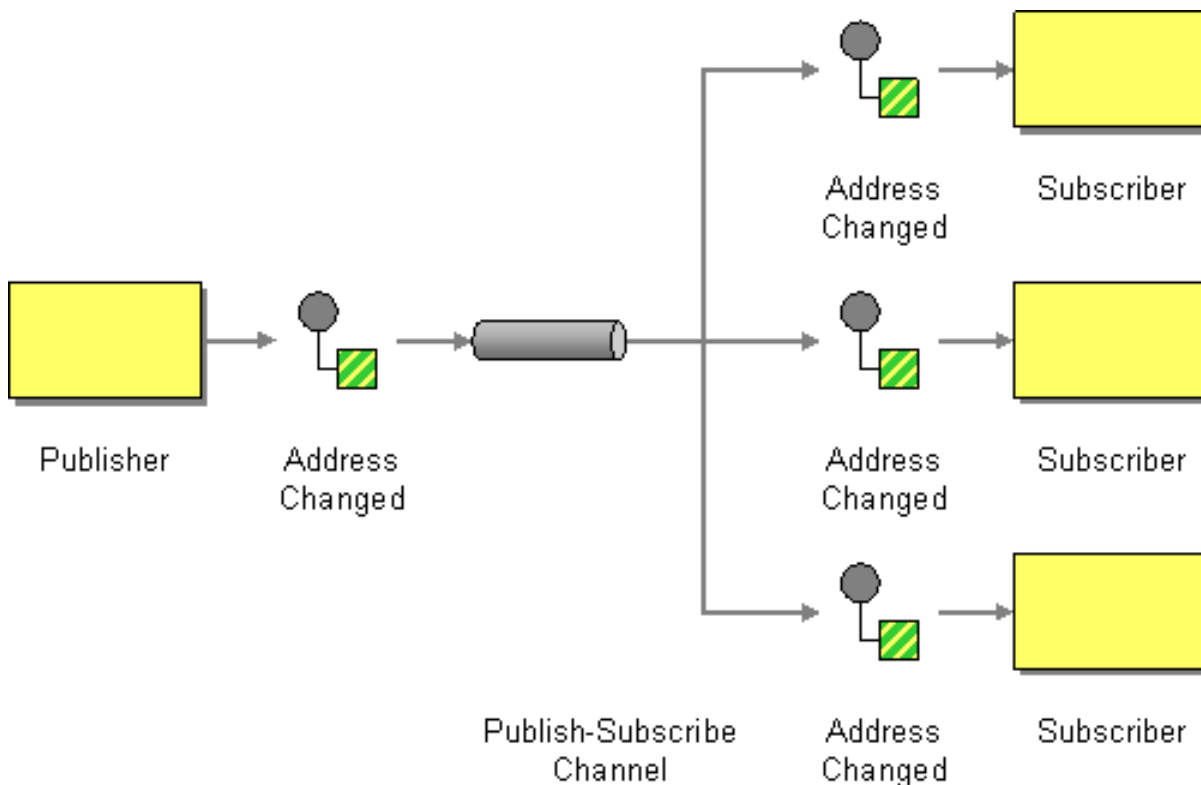
To configure an [Endpoint](#) to use a custom `PollingConsumerPollStrategy` you use the option `pollStrategy`. For example in the file consumer below we want to use our custom strategy defined in the [Registry](#) with the bean id `myPoll`:

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

2.39. Publish Subscribe Channel

Camel supports the [Publish Subscribe Channel](#) from the EIP patterns using the following components

- [JMS](#) for working with JMS Topics for high performance, clustering and load balancing
- [XMPP](#) when using rooms for group communication



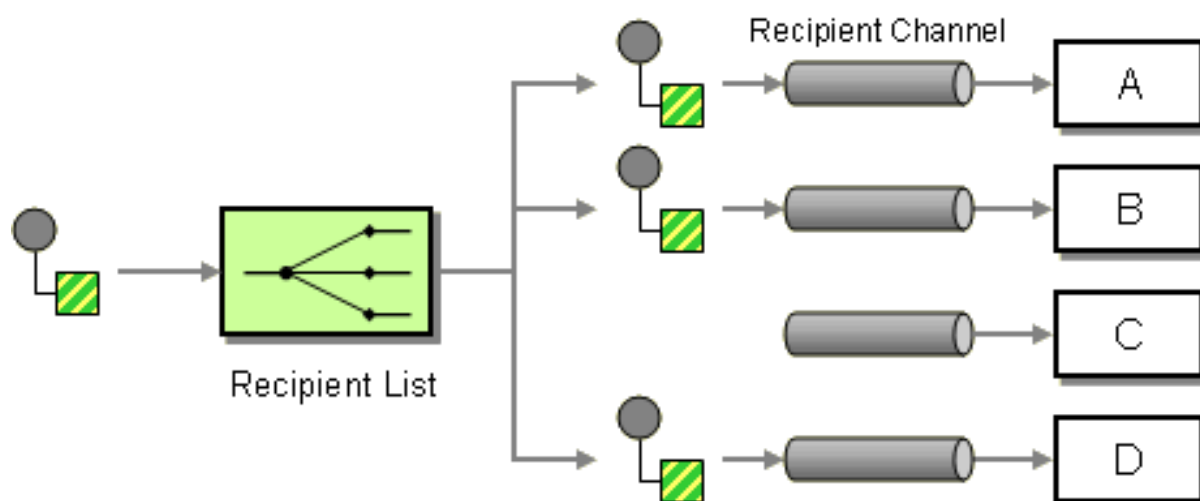
Another option is to explicitly list the publish-subscribe relationship using routing logic; this keeps the producer and consumer decoupled but lets you control the fine grained routing configuration using the [DSL](#) or [XML Configuration](#).

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>
```

2.40. 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.

2.40.1. Options

Name	Default Value	Description
delimiter	,	Delimiter used if the Expression returned multiple endpoints. <i>Camel 2.13</i> can be disabled using "false"
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. From Camel 2.12 onwards you can also use a POJO as the AggregationStrategy , see the Aggregate page for more details.

Name	Default Value	Description
strategyMethodName		Camel 2.12: This option can be used to explicit declare the method name to use, when using POJOs as the <code>AggregationStrategy</code> . See the Aggregate page for more details.
strategyMethodAllowNull	false	Camel 2.12: If this option is <code>false</code> then the aggregate method is not used if there was no data to enrich. If this option is <code>true</code> then null values is used as the <code>oldExchange</code> (when no data to enrich), when using POJOs as the <code>AggregationStrategy</code> . See the Aggregate page for more details.
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.
parallelAggregate	false	Camel 2.14: If enabled then the aggregate method on <code>AggregationStrategy</code> can be called concurrently. Notice that this would require the implementation of <code>AggregationStrategy</code> to be implemented as thread-safe. By default this is <code>false</code> meaning that Camel synchronizes the call to the aggregate method. Though in some use-cases this can be used to archive higher performance when the <code>AggregationStrategy</code> is implemented as thread-safe.
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 and the Recipient List breaks out, with message flow continuing to the next element. Note that if you provide a TimeoutAwareAggregationStrategy , its <code>onTimeout()</code> method is invoked before breaking out. If the timeout is reached with running tasks still remaining, certain tasks for which it is difficult for Camel to shut down in a graceful manner may continue to run. So use this option with a bit of care.
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.
cacheSize	1000	Camel 2.13.1/2.12.4: Allows to configure the cache size for the <code>ProducerCache</code> which caches producers for reuse in the routing slip. Will by default use the default cache size which is 1000. Setting the value to -1 allows to turn off the cache all together.

2.40.2. 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>
```

2.40.3. 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(", "));
```

2.40.3.1. Iterable value

The dynamic list of recipients that are defined in the header must be iterable 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 see this [JUnit test case](#).

2.40.3.2. 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:

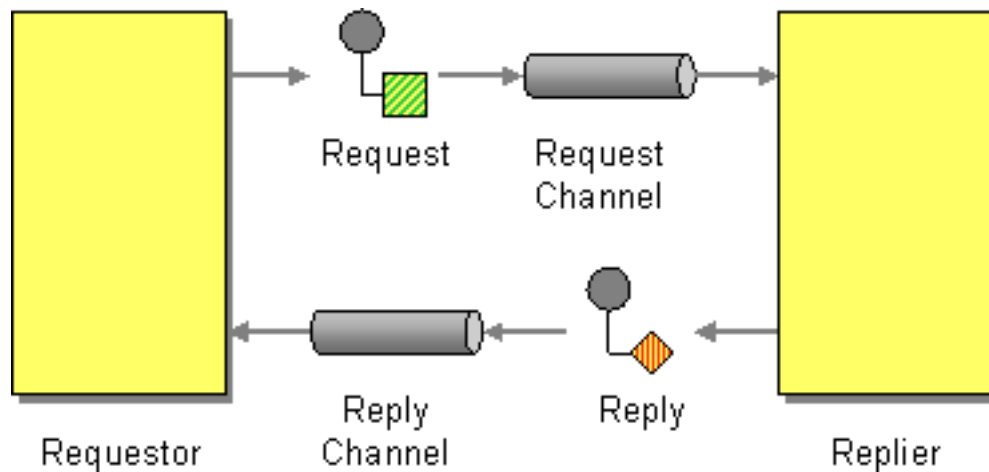
```
from("direct:a").recipientList(header("myHeader"), "#");
```

2.41. Request Reply

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



See also the related [Event Message](#) EIP.



For example when using [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).

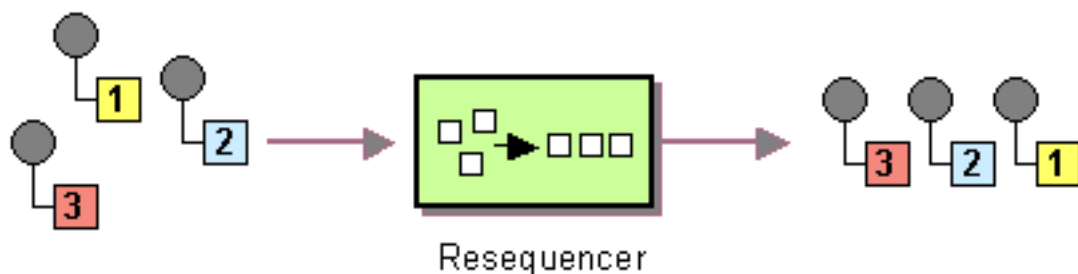
When consuming messages from [JMS](#) a Request-Reply is indicated by the presence of the **JMSReplyTo** header. You can explicitly force an endpoint to be in Request Reply mode by setting the exchange pattern on the URI. e.g.

```
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.

2.42. 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 *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.

2.42.1. 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" />
    </resequence>
  </route>
</camelContext>
```

```

<!--
    batch-config can be omitted for default (batch)
    resequencer settings
-->
<batch-config batchSize="300" batchTimeout="4000" />
</resequence>
</route>
</camelContext>

```

In the batch mode, you can also 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 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.

2.42.2. 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 successor 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>

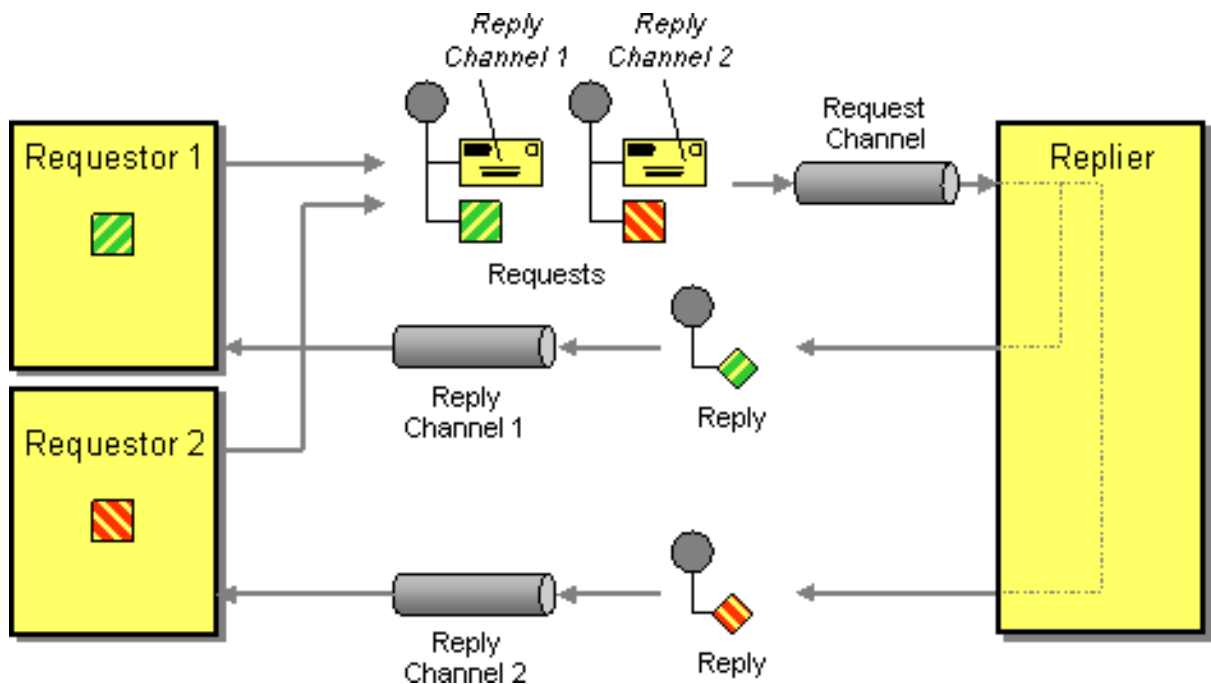
```

2.42.3. Further Examples

See the [Camel Website](#) for further examples of this component in use.

2.43. Return Address

Camel supports the [Return Address](#) from the EIP patterns by using the `JMSReplyTo` header.



For example when using [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>

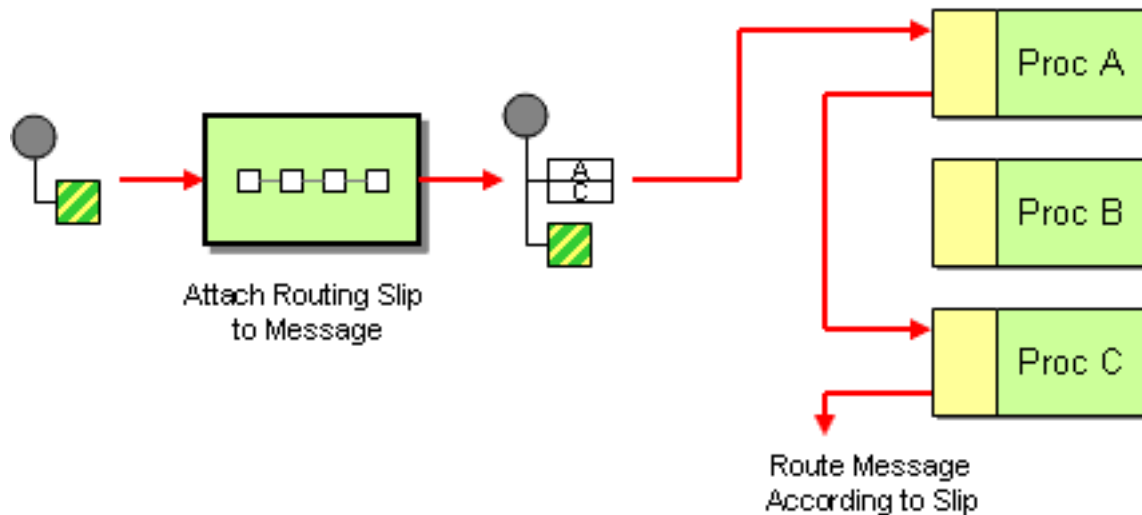
<route>
<from uri="activemq:queue:foo"/>
<transform>
<simple>Bye ${in.body}</simple>
</transform>
</route>
```

```
<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](#).

2.44. 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.



2.44.1. Options

Name	Default Value	Description
uriDelimiter	,	Delimiter used if the Expression returned multiple endpoints.
ignoreInvalidEndpoints	false	If an endpoint uri could not be resolved, should it be ignored. Otherwise Camel will throw an exception stating the endpoint uri is not valid.
cacheSize	1000	Camel 2.13.1/2.12.4: Allows to configure the cache size for the <code>ProducerCache</code> which caches producers for reuse in the routing slip. Will by default use the default cache size which is 1000. Setting the value to -1 allows to turn off the cache all together.

2.44.2. 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("aRoutingSlipHeader");
```

Messages will be checked for the existence of the "aRoutingSlipHeader" header. The value of this header should be a comma-delimited list of endpoint [URIs](#) you wish the message to be routed to. The [Message](#) will be routed in a [pipeline](#) fashion (i.e. one after the other).

The [Routing Slip](#) will set a property (`Exchange.SLIP_ENDPOINT`) on the [Exchange](#) which contains the current endpoint as it advanced through the slip. This allows you to *know* how far we have processed in the slip.

The [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 [Dynamic Router](#) pattern instead.

For further examples of this pattern in use see the Camel [routing slip test cases](#).

2.44.3. 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 uriDelimiter="#">
      <header>aRoutingSlipHeader</header>
    </routingSlip>
  </route>
</camelContext>
```

2.44.4. Ignore invalid endpoints

The [Routing Slip](#) now supports `ignoreInvalidEndpoints` which the [Recipient List](#) also supports. You can use it to skip endpoints which are 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 ignoreInvalidEndpoints="true">
    <header>myHeader</header>
  </routingSlip>
</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.

2.44.5. Expression supporting

The [Routing Slip](#) now supports to take the expression parameter as the [Recipient List](#) does. You can tell 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>
```

2.45. 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.

You can use this EIP with the `sample` DSL as shown in the following examples:

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"/>
```



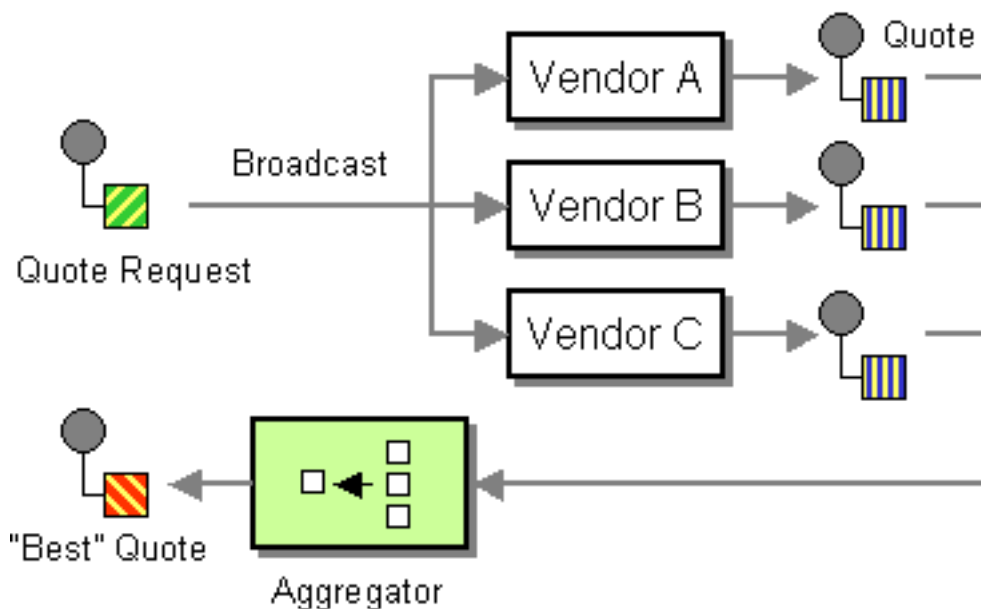
```
<sample messageFrequency="5">
  <to uri="mock:result"/>
</sample>
</route>
```

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>
```

2.46. 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.



2.46.1. Dynamic Scatter-Gather Example

In this example we want to get the best quote for beer from several different vendors. We use a dynamic [Recipient List](#) to get the request for a quote to all vendors and an [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>
</camelContext>
```

```

<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 *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 [Aggregator](#) with a custom aggregation strategy. The [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](#)

2.46.2. Static Scatter-Gather Example

You can lock down which recipients are used in the Scatter-Gather by using a static [Recipient List](#). It looks something like this

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

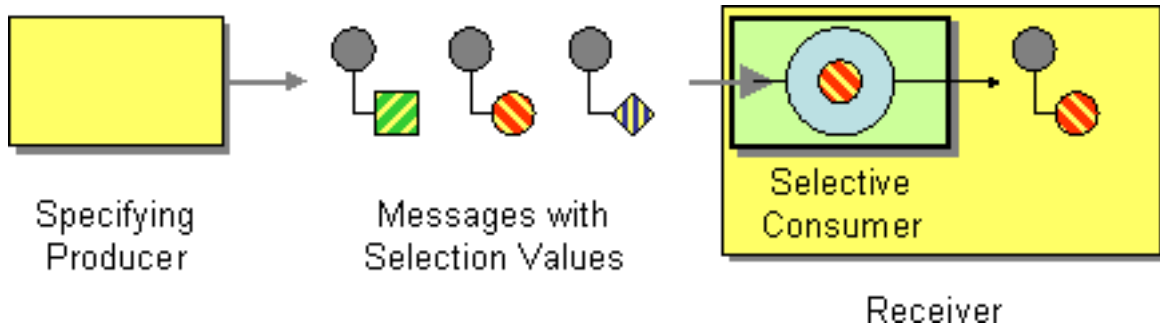
from("seda:vendor1").to("bean:vendor1").to("seda:quoteAggregator");
from("seda:vendor2").to("bean:vendor2").to("seda:quoteAggregator");
from("seda:vendor3").to("bean:vendor3").to("seda:quoteAggregator");

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

```
"mock:result")
```

2.47. 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 [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 [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);
    }
};
```

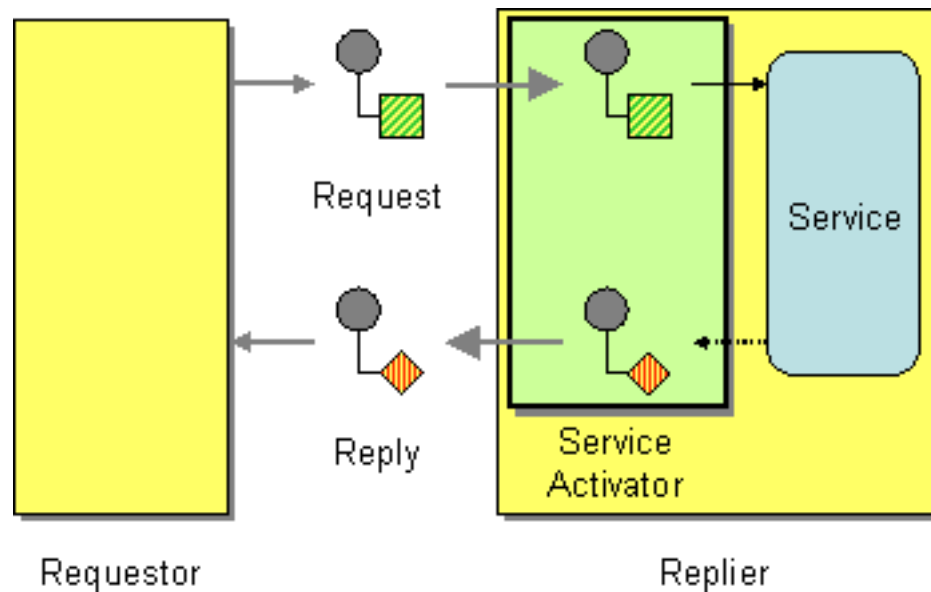
Using the [Spring XML Extensions](#)

```
<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>
```

2.48. Service Activator

Camel has several endpoint components that support the [Service Activator](#) from the EIP patterns.



Components like [Bean](#), [CXF](#) and [Pojo](#) provide 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>
```

2.49. 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 <code>java.util.Comparator</code> to use for sorting the message body. Camel will by default use a comparator which does a A..Z sorting.

2.49.1. Java DSL Example

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");
```

2.49.2. Spring DSL Example

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

Example 2.1.

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

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

Example 2.2.

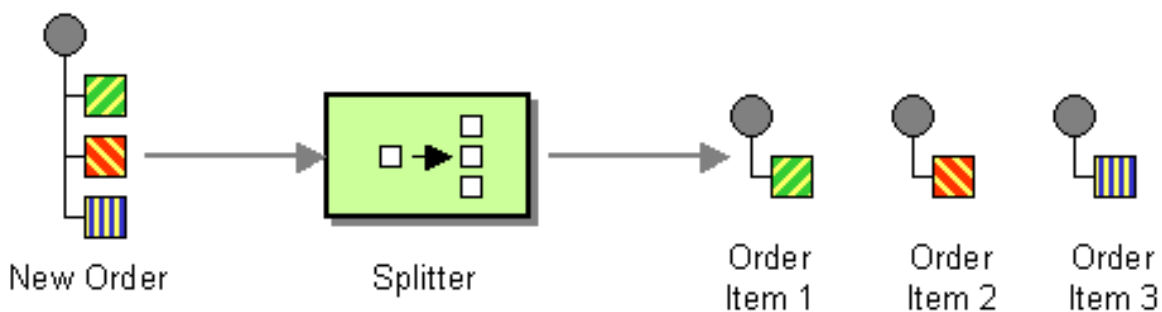
```
<route>
  <from uri="file://inbox"/>
  <sort comparatorRef="myReverseComparator">
    <simple>body</simple>
  </sort>
  <beanRef ref="MyServiceBean" method="processLine"/>
</route>

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

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

2.50. Splitter

The [Splitter](#) from the EIP patterns allows you to 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. From Camel 2.12 onwards you can also use a POJO as the AggregationStrategy, see the Aggregate page for more details.
strategyMethodName		Camel 2.12: This option can be used to explicit declare the method name to use, when using POJOs as the AggregationStrategy. See the Aggregate page for more details.
strategyMethodAllowNull	false	Camel 2.12: If this option is <i>false</i> then the aggregate method is not used for the very first splitted message. If this option is <i>true</i> then null values is used as the <code>oldExchange</code> (for the very first message splitted), when using POJOs as the AggregationStrategy. See the Aggregate page for more details.
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.
parallelAggregate	false	Camel 2.14: If enabled then the aggregate method on AggregationStrategy can be called concurrently. Notice that this would require the implementation of AggregationStrategy to be implemented as thread-safe. By default this is <i>false</i> meaning that Camel synchronizes the call to the aggregate method. Though in some use-cases this can be used to archive higher performance when the AggregationStrategy is implemented as thread-safe.
executorServiceRef		Refers to a custom Thread Pool to be used for parallel processing. Notice if you set this option, then parallel processing is automatically 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 disable, 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		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. If the timeout is reached with running tasks still remaining, certain tasks for which it is difficult for Camel to shut down in a graceful manner may continue to run. So use this option with a bit of care.
onPrepareRef		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	Whether the unit of work should be shared. See further below for more details.

Exchange Properties:

Property	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. 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.



The [Splitter](#) will by default return the **last** splitted message.

The [Splitter](#) will by default return the original input message.

For all versions You can override this by supplying your own strategy as an `AggregationStrategy`. See the Camel Website for the [split aggregate request/reply sample](#). It uses the same strategy the [Aggregator](#) supports. This [Splitter](#) can be viewed as having a build in light weight [Aggregator](#).

2.50.1. 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 see this [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>

```

2.50.2. 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.

2.50.3. Splitting a Collection, Iterator or Array

A common use case is to split a Collection, Iterator or Array from the message. In the sample below we simply use an [Expression](#) to identify the value to split.

```

from("direct:splitUsingBody").split(body()).to("mock:result");
from("direct:splitUsingHeader").split(header("foo")).to("mock:result");

```

In Spring XML you can use the [Simple](#) language to identify the value to split.

```

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

<split>
  <simple>${header.foo}</simple>
  <to uri="mock:result"/>
</split>

```

2.50.4. 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");
```

2.50.5. 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. Alternatively, you can use the Tokenizer language which supports this when you supply the start and end tokens. From Camel 2.14, you can use the XMLTokenizer language which is specifically provided for tokenizing XML documents.

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")
```

2.50.6. Streaming big XML payloads using Tokenizer language

There are two tokenizers that can be used to tokenize an XML payload. The first tokenizer uses the same principle as in the text tokenizer to scan the XML payload and extract a sequence of tokens.

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.

2.50.7. Specifying a custom aggregation strategy

This is specified similar to the [Aggregator](#).

2.50.8. 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");
```

2.50.9. Using a Pojo to do the splitting

As the *Splitter* can use any *Expression* to do the actual splitting we leverage this fact and use a **method** expression to invoke a *Bean* to get the splitted parts. The *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 *Bean* that we have registered with the id `mySplitterBean` in the *Registry*.

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

And the logic for our *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 iterable
     * 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
     * @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;
}
```

2.50.10. Stop processing in case of exceptions

The [Splitter](#) will by default continue to process the entire [Exchange](#) even in case of one of the splitted message will throw 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>
```

2.50.11. 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");
```

2.51. 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 as a positive number. Note, in the XML DSL, this option is configured using an Expression instead of an attribute.
timePeriodMillis	1000	The time period in milliseconds, 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.
rejectExecution	false	Camel 2.14: If this option is true, throttler throws a ThrottlerRejectExecutionException when the request rate exceeds the limit.

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 since `timePeriodMillis` defaults to 1000 milliseconds, just setting the `maximumRequestsPerPeriod` has the effect of setting the maximum number of requests per 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 see this [JUnit test case](#).

Using the [Spring XML Extensions](#)

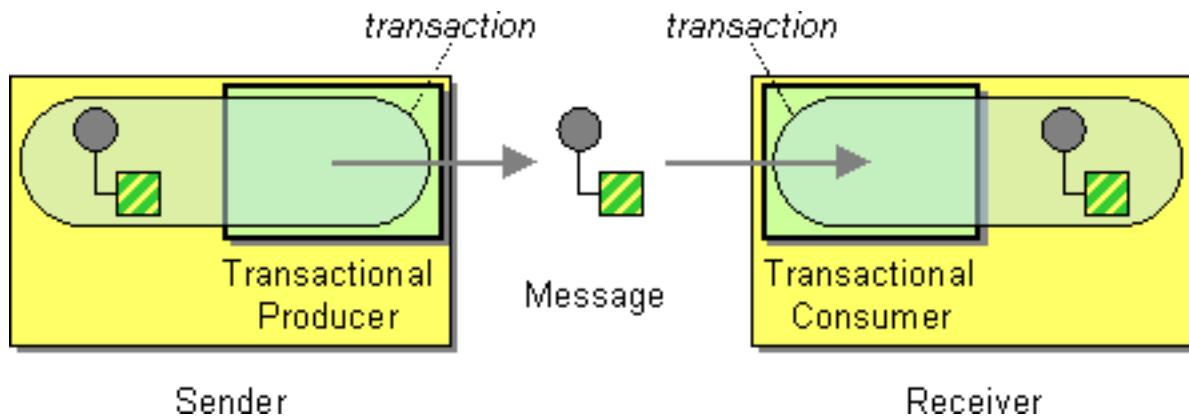
```
<route>
  <from uri="seda:a" />
  <throttle timePeriodMillis="10000"/>
    <constant>3</constant>
    <to uri="mock:result" />
  </throttle>
</route>
```

You can let the [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");
```

2.52. Transactional Client

Camel recommends supporting the [Transactional Client](#) from the EIP patterns using Spring transactions.



Transaction Oriented Endpoints ([Camel Toes](#)) like *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.



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 `TransactionTemplates` 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.

For more information on the `TransactionTemplate` and `PlatformTransactionManager` classes, see the Spring Framework API documentation on <http://spring.io/docs>.

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);
```

2.52.1. 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 required = 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");
```

2.52.2. 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>
```

2.52.3. 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 TransactionManager, 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 -->
<!-- that 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.

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



```

        </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.

2.52.4. 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.

```

<!-- 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>

<!-- 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.

```

<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"/>

```



Which 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.

2.52.5. Integration Testing with Spring



An Integration Test here means a test runner class annotated `@RunWith(SpringJUnit4ClassRunner.class)`.

When following the Spring Transactions documentation it is tempting to annotate your integration test with `@Transactional` then seed your database before firing up the route to be tested and sending a message in. This is incorrect as Spring will have an in-progress transaction, and Camel will wait on this before proceeding, leading to the route timing out.

Instead, remove the `@Transactional` annotation from the test method and seed the test data within a `TransactionTemplate` execution which will ensure the data is committed to the database before Camel attempts to pick up and use the transaction manager. A simple example [can be found on GitHub](#).

Spring's transactional model ensures each transaction is bound to one thread. A Camel route may invoke additional threads which is where the blockage may occur. This is not a fault of Camel but as the programmer you must be aware of the consequences of beginning a transaction in a test thread and expecting a separate thread created by your Camel route to be participate, which it cannot. You can, in your test, mock the parts that cause separate threads to avoid this issue.

2.53. 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.

2.53.1. 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");
```

2.53.2. 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"/>
```

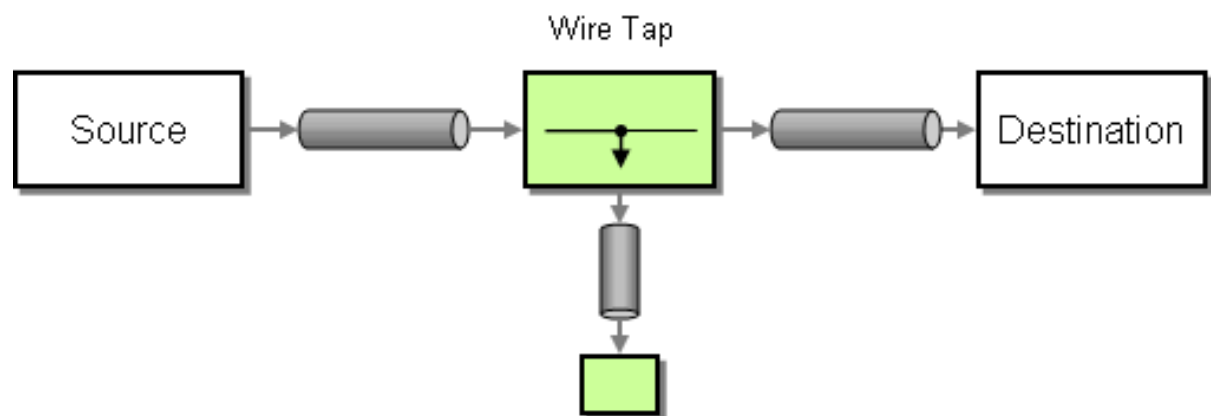
The XML DSL to validate the message header would look like this:

```
<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"/>
```

2.54. 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.



If you [Wire Tap](#) a stream message body then you should consider enabling [Stream caching](#) to ensure the message body can be read at each endpoint. See more details at [Stream caching](#).

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.

Name	Default Value	Description
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.

2.54.1. 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. Beware that only the Exchange is copied - Wire Tap won't do a deep clone (unless you specify a custom processor via onPrepareRef which does that). So all copies could share objects from the original Exchange.
- 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.

2.54.2. 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
ActiveMQ / activemq-camel activemq:[topic:]destinationName	For JMS Messaging with Apache ActiveMQ
AHC / camel-ahc ahc:http[s]://hostname[:port][/resourceUri][?options]	To call external HTTP services using Async Http Client
Atom / camel-atom atom:uri	Working with Apache Abdera for atom integration, such as consuming an atom feed.
Apns / camel-apns apns:<notify consumer>[?options]	For sending notifications to Apple iOS devices
Avro / camel-avro avro:[transport]:[host]:[port][/messageName][?options]	Working with Apache Avro for data serialization.
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).
Cache / camel-cache cache://cachename[?options]	The cache component facilitates creation of caching endpoints and processors using EHCache as the cache implementation.
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).
CMIS / camel-cmis cmis://cmisServerUrl[?options]	Uses the Apache Chemistry client API to interface with CMIS supporting CMS
Context / camel-context	Used to refer to endpoints within a separate CamelContext to provide a simple black box composition approach so that routes

Component / ArtifactId / URI	Description
context:camelContextId: localEndpointName	can be combined into a CamelContext and then used as a black box component inside other routes in other CamelContexts
<i>CouchDB</i> / camel-couchdb couchdb:hostName[:port]/database[?options]	To integrate with Apache CouchDB .
<i>Crypto (Digital Signatures)</i> crypto:sign:name[?options], crypto:verify:name[?options]	Used to sign and verify exchanges using the Signature Service of the Java Cryptographic Extension.
<i>CXF</i> / camel-cxf cxf:address[?serviceClass=...]	Working with Apache CXF for web services integration
<i>CXF Bean Component</i> / 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.
<i>CXFRS</i> / camel-cxf cxfrs:address[?resourcesClasses=...]	Working with Apache CXF for REST services integration
<i>Direct</i> / camel-core direct:name	Synchronous call to another endpoint from same CamelContext
<i>Disruptor</i> / camel-disruptor disruptor:someName[?<option>] disruptor-vm:someName[?<option>]	To provide the implementation of SEDA which is based on disruptor
<i>ElasticSearch</i> / camel-elasticsearch elasticsearch://clusterName[?options]	For interfacing with an ElasticSearch server.
<i>Spring Event</i> / camel-spring spring-event://default[?options]	Working with Spring ApplicationEvents
<i>Exec</i> / camel-exec exec://executable[?options]	For executing system commands
<i>Facebook</i> / camel-facebook facebook://endpoint[?options]	Providing access to all of the Facebook APIs accessible using Facebook4J
<i>File</i> / camel-core file://nameOfFileOrDirectory	Sending messages to a file or polling a file or directory.
<i>Flatpack</i> / camel-flatpack flatpack:[fixed delim]:configFile	Processing fixed width or delimited files or messages using the FlatPack library
<i>FOP</i> / camel-fop fop:outputFormat[?options]	Renders the message into different output formats using Apache FOP
<i>Freemarker</i> / camel-freemarker freemarker:someTemplateResource	Generates a response using a Freemarker template
<i>FTP</i> / camel-ftp ftp://host[:port]/fileName	Sending and receiving files over FTP.
<i>FTP</i> / camel-ftp (FTPS) ftps://host[:port]/fileName	Sending and receiving files over FTP Secure (TLS and SSL).
<i>Geocoder</i> / camel-geocoder geocoder:<address latlng:latitude,longitude>[?options]	Supports looking up geocoders for an address, or reverse lookup geocoders from an address.

Component / ArtifactId / URI	Description
Guava EventBus / camel-guava-eventbus guava-eventbus:busName[?options]	The Google Guava EventBus allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other). This component provides integration bridge between Camel and Google Guava EventBus infrastructure.
HDFS / camel-hdfs hdfs://hostName[:port][/path][?options]	For reading/writing from/to an HDFS filesystem using Hadoop 1.x
HDFS2 / camel-hdfs2 hdfs2://hostName[:port][/path][?options]	For reading/writing from/to an HDFS filesystem using Hadoop 2.x
HL7 mina:tcp://hostname[:port]	For working with the HL7 MLLP protocol and the HL7 model using the HAPI library.
HTTP4 / camel-http4 http4://hostname[:port]	For calling out to external HTTP servers using Apache HTTP Client 4.x
Infinispan / camel-infinispan infinispan://hostName[?options]	For reading/writing from/to Infinispan distributed key/value store and data grid
Mail / camel-mail imap://hostname[:port]	Receiving email using IMAP
Jasypt / camel-jasypt jasypt: uri	Simplified on-the-fly encryption library, integrated with Camel.
JCR / camel-jcr jcr://user:password@repository/path/to/node	Storing a message in a JCR (JSR-170) compliant repository like Apache Jackrabbit
JDBC / camel-jdbc jdbc:dataSourceName?options	For performing JDBC queries and operations
Jetty / camel-jetty jetty:url	For exposing services over HTTP
JGroups / camel-jgroups jgroups:clusterName[?options]	The jgroups: component provides exchange of messages between Camel infrastructure and JGroups clusters.
JMS / camel-jms jms:[topic:]destinationName	Working with JMS providers
JMX / camel-jmx jmx://platform?options	For working with JMX notification listeners
JPA / camel-jpa jpa://entityName	For using a database as a queue via the JPA specification for working with OpenJPA , Hibernate or TopLink
Jsch / camel-jsch scp://localhost/destination	Support for the scp protocol.
Kafka / camel-kafka kafka://server:port[?options]	For producing to or consuming from Apache Kafka message brokers.
Krati / camel-krati krati://[path to datastore/][?options]	For producing to or consuming to Krati datastores
LDAP / camel-ldap ldap:host[:port]?base=... [&scope=<scope>]	Performing searches on LDAP servers (<scope> must be one of object onelevel subtree)
Log / camel-core	Uses Jakarta Commons Logging to log the message exchange to some underlying logging system like log4j

Component / ArtifactId / URI	Description
log:loggingCategory[?level=ERROR]	
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
Mail / camel-mail mail://user-info@host:port	Sending and receiving email
Mock / camel-core mock:name	For testing routes and mediation rules using mocks
Mail / camel-mail pop3://user-info@host:port	Receiving email using POP3 and JavaMail
MINA 2 / camel-mina2 mina2:[tcp udp vm]:host[:port][?options]	Working with Apache MINA 2.x
MongoDB / camel-mongodb mongodb:connectionBean[?options]	Interacts with MongoDB databases and collections. Offers producer endpoints to perform CRUD-style operations and more against databases and collections, as well as consumer endpoints to listen on collections and dispatch objects to Camel routes
MQTT / camel-mqtt mqtt:name[?options]	Component for communicating with MQTT M2M message brokers
Mustache / camel-mustache mustache:templateName[?options]	Generates a response using a Mustache template
MyBatis / camel-mybatis mybatis://statementName	Performs a query, poll, insert, update or delete in a relational database using MyBatis
OptaPlanner / camel-optaplanner optaplanner:solverConfig[?options]	Solves the planning problem contained in a message with OptaPlanner .
Properties / camel-core properties://key[?options]	The properties component facilitates using property placeholders directly in endpoint uri definitions.
Quartz / camel-quartz quartz://groupName/timerName	Provides a scheduled delivery of messages using the Quartz scheduler
Quartz2 / camel-quartz2 quartz2://groupName/timerName[?options]	Provides a scheduled delivery of messages using the Quartz 2.x scheduler
Ref / camel-core ref:name	Component for lookup of existing endpoints bound in the Camel Registry.
RabbitMQ Component / camel-rabbitmq rabbitmq://hostname[:port]/exchangeName[?options]	Component for integrating with RabbitMQ
RMI / camel-rmi rmi://host[:port]	Working with RMI
RSS / camel-rss rss:uri	Working with ROME for RSS integration, such as consuming an RSS feed.
Salesforce / camel-salesforce salesforce:topic[?options]	Working with ROME for RSS integration, such as consuming an RSS feed.
SAP NetWeaver / camel-sap-netweaver sap-netweaver:hostName[:port][?options]	To integrate with Salesforce
SEDA / camel-core	Asynchronous call to another endpoint in the same Camel Context

Component / ArtifactId / URI	Description
<code>seda:name</code>	
Servlet / camel-servlet <code>servlet:uri</code>	For exposing services over HTTP through the servlet which is deployed into the Web container.
FTP / camel-ftp (SFTP) <code>sftp://host[:port]/fileName</code>	Sending and receiving files over SFTP (FTP over SSH).
Mail / camel-mail <code>smtp://user-info@host[:port]</code>	Sending email using SMTP and JavaMail
SJMS / camel-sjms <code>sjms:[queue: topic:]destinationName[?options]</code>	A ground up implementation of a JMS client
SMPP / camel-smpp <code>smpp://user-info@host[:port]?options</code>	To send and receive SMS using Short Messaging Service Center using the JSMPP library
SNMP / camel-snmp <code>snmp://host[:port]?options</code>	Polling OID values and receiving traps using SNMP via SNMP4J library
Solr / camel-solr <code>solr://hostName[:port]/solr[?options]</code>	Uses the Solrj client API to interface with an Apache Lucene Solr server
Splunk / camel-splunk <code>splunk://[endpoint]?[options]</code>	For working with Splunk
Spring Batch / camel-spring-batch <code>spring-batch:jobName[?options]</code>	To bridge Camel and Spring Batch
Spring Integration / camel-spring-integration <code>spring-integration: defaultChannelName</code>	The bridge component of Camel and Spring Integration
Spring LDAP / camel-spring-ldap <code>spring-ldap:springLdapTemplateBean[?options]</code>	Camel wrapper for Spring LDAP
Spring Redis / camel-spring-redis <code>spring-redis://hostName:port[?options]</code>	Component for consuming and producing from Redis key-value store Redis
Spring Web Services / camel-spring-ws <code>spring-ws:[mapping-type:]address[?options]</code>	Client-side support for accessing web services, and server-side support for creating your own contract-first web services using Spring Web Services
SQL Component / camel-sql <code>sql:select * from table where id=#</code>	Performing SQL queries using JDBC
SSH / camel-ssh <code>ssh:[username[:password]@]host[:port][?options]</code>	For sending commands to a SSH server
StAX / camel-stax <code>stax:(contentHandlerClassName #myHandler)</code>	Process messages through a SAX ContentHandler .
Stomp / camel-stomp <code>stomp:queue:destinationName[?options]</code>	For communicating with Stomp compliant message brokers, like Apache ActiveMQ or ActiveMQ Apollo
Stub <code>stub:someOtherCamelUri</code>	Allows you to stub out some physical middleware endpoint for easier testing or debugging
Test / camel-spring <code>test:expectedMessagesEndpointUri</code>	Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint
Timer / camel-core <code>timer://name</code>	A timer endpoint

Component / ArtifactId / URI	Description
Twitter / camel-twitter twitter://endpoint[?options]	A twitter endpoint
Velocity / camel-vertx vertx:eventBusName	Working with the vertx event bus
Vertex / camel-velocity velocity:someTemplateResource	Generates a response using an Apache Velocity template
VM / camel-core vm:name	Asynchronous call to another endpoint in the same JVM
Weather / camel-weather wweather://name[?options]	Polls the weather information from Open Weather Map
Websocket / camel-websocket websocket://hostname[:port][/resourceUri][?options]	Communicating with Websocket clients
XQuery Endpoint / camel-saxon xquery:someXQueryResource	Generates a response using an XQuery template
XSLT / camel-spring xslt:someTemplateResource	Generates a response using an XSLT template
Yammer / camel-yammer yammer://function[?options]	Allows you to interact with the Yammer enterprise social network
Zookeeper camel-zookeeper zookeeper://zookeeperServer[:port][/path][?options]	Working with ZooKeeper cluster(s)

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 [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`.

Transacted and caching

See section *Transactions and Cache Levels* below on [JMS](#) page if you are using transactions with [JMS](#) as it can impact performance.

3.1.1. URI format and Options

```
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
```

For options, see [JMS](#) component as all these options also apply for this component.

3.1.2. 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"));
```

3.1.3. 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>
```

3.1.4. 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>5.6.0</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"
      init-method="start" destroy-method="stop">
  <property name="maxConnections" value="8" />
  <property name="connectionFactory" ref="jmsConnectionFactory" />
</bean>

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

<bean id="activemq"
      class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="configuration" ref="jmsConfig"/>

  <!-- if we are using transacted then enable CACHE_CONSUMER (if not using XA) to run
  faster
  see more details at: http://camel.apache.org/jms
  <property name="transacted" value="true"/>
  <property name="cacheLevelName" value="CACHE_CONSUMER" />
  -->
</bean>
```

Notice the `init` and `destroy` methods on the pooled connection factory. This is important to ensure the connection pool is properly started and shutdown.

If you are using transactions then see more details at [JMS](#). And remember to set `cacheLevelName` to `CACHE_CONSUMER` if you are not using XA transactions. This can dramatically improve performance.

The `PooledConnectionFactory` will then create a connection pool with up to 8 connections in use at the same time. Each connection can be shared by many sessions. There is an option `maxActive` you can use to configure the maximum number of sessions per connection; the default value is 500. From **ActiveMQ 5.7** onwards the option has been renamed to `maxActiveSessionPerConnection` to better reflect its purpose. Notice the `concurrentConsumers` is set to a higher value than `maxConnections` is. This is acceptable because each consumer uses a session and sessions can share the same connection. In the above example we can have $8 * 500 = 4000$ active simultaneous sessions.

3.1.5. 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.

3.1.6. Using ActiveMQ Destination Options

Available as of ActiveMQ 5.6

You can configure the [Destination Options](#) in the endpoint uri, using the "destination." prefix. For example to mark a consumer as exclusive, and set its prefetch size to 50, you can do as follows:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="file://src/test/data?noop=true"/>
      <to uri="activemq:queue:foo"/>
    </route>
    <route>
      <!-- use consumer.exclusive ActiveMQ destination option, notice we have to prefix
      with destination. -->
      <from uri="activemq:foo?
      destination.consumer.exclusive=true&destination.consumer.prefetchSize=50"/>
      <to uri="mock:results"/>
    </route>
  </camelContext>
```

```
</camelContext>
```

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>${in.body}&#13;</simple>
  </transform>
  <to uri="file://data/activemq/?fileExist=Append&
    fileName=advisoryConnection-${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 this dependency

- `activemq-camel`

[ActiveMQ](#) is an extension of the [JMS](#) component released with the [ActiveMQ project](#).

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-camel</artifactId>
  <version>5.6.0</version>
</dependency>
```

3.2. AHC

The AHC component provides HTTP based endpoints for consuming external HTTP resources (as a client to call external servers using HTTP).

The component uses the [Async Http Client](#) library.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ahc</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.2.1. URI format and Options

```
ahc:http://hostname[:port][/resourceUri][?options]
ahc:https://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&...`, where *option* can be:

Table 3.1. AhcEndpoint Options

Option	Value	Behavior
throwExceptionOnFailure	true	Option to disable throwing the <code>AhcOperationFailedException</code> in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.
bridgeEndpoint	false	The entire list of entries from the feed is set: If the option is true, then the <code>Exchange.HTTP_URI</code> header is ignored, and use the endpoint's URI for request. You may also set the <code>throwExcpetionOnFailure</code> to be false to let the <code>AhcProducer</code> send all the fault response back.
transferException	false	If enabled and an exchange failed processing on the consumer side, and if the caused exception was sent back serialized in the response as a <code>application/x-java-serialized-object</code> content type (for example using Jetty or <code>SERVLET</code> Camel components). On the producer side the exception will be deserialized and thrown as is, instead of the <code>AhcOperationFailedException</code> . The caused exception is required to be serialized.
client	null	To use a custom <code>com.ning.http.client.AsyncHttpClient</code> .
clientConfig	null	To configure the <code>AsyncHttpClient</code> to use a custom <code>com.ning.http.client.AsyncHttpClientConfig</code> instance. This instance replaces any instance configured at the component level.

Option	Value	Behavior
<code>clientConfig.x</code>	null	To configure additional properties of the <code>com.ning.http.client.AsyncHttpClientConfig</code> instance used by the endpoint. Note that configuration options set using this parameter will be merged with those set using the <code>clientConfig</code> parameter or the instance set at the component level with properties set using this parameter taking priority.
<code>clientConfig.realm.x</code>	null	Starting with Camel 2.11, to configure realm properties of the <code>com.ning.http.client.AsyncHttpClientConfig</code> . The options can be used are the options from <code>com.ning.http.client.Realm.RealmBuilder..</code> For example, to set scheme, you can configure <code>clientConfig.realm.scheme=DIGEST</code> .
<code>binding</code>	null	To use a custom <code>org.apache.camel.component.ahc.AhcBinding</code> .
<code>sslContextParameters</code>	null	Starting with Camel 2.9, reference to a <code>org.apache.camel.util.jsse.SSLContext</code> Parameters in the Registry. This reference overrides any configured <code>SSLContextParameters</code> at the component level. See Using the JSSE Configuration Utility . Note that configuring this option will override any SSL/TLS configuration options provided through the <code>clientConfig</code> option at the endpoint or component level.
<code>bufferSize</code>	4096	Starting with Camel 2.10.3, the initial in-memory buffer size used when transferring data between Camel and AHC Client.

Table 3.2. AhcComponent Options

Option	Value	Behavior
<code>client</code>	null	To use a custom <code>com.ning.http.client.AsyncHttpClient</code> .
<code>clientConfig</code>	null	To configure the <code>AsyncHttpClient</code> to use a custom <code>com.ning.http.client.AsyncHttpClientConfig</code> .
<code>binding</code>	null	To use a custom <code>org.apache.camel.component.ahc.AhcBinding</code> .
<code>sslContextParameters</code>	null	Starting with Camel 2.9, to configure custom SSL/TLS configuration options at the component level. See Using the JSSE Configuration Utility for more details. Note that configuring this option will override any SSL/TLS configuration options provided through the <code>clientConfig</code> option at the endpoint or component level.

3.2.2. Message Headers

Camel AHC uses these headers:

Header	Type	Description
<code>Exchange.HTTP_URI</code>	String	URI to call. Will override existing URI set directly on the endpoint.
<code>Exchange.HTTP_PATH</code>	String	Request URI's path, the header will be used to build the request URI with the <code>HTTP_URI</code> . If the path is start with "/", http producer will try to find the relative path based on the <code>Exchange.HTTP_BASE_URI</code> header or the <code>exchange.getFromEndpoint().getEndpointUri()</code> .
<code>Exchange.HTTP_QUERY</code>	String	URI parameters. Will override existing URI parameters set directly on the endpoint.
<code>Exchange.HTTP_RESPONSE_CODE</code>	int	The HTTP response code from the external server. Is 200 for OK.
<code>Exchange.HTTP_CHARACTER_ENCODING</code>	String	Character encoding.
<code>Exchange.CONTENT_TYPE</code>	String	The HTTP content type. Is set on both the IN and OUT message to provide a content type, such as <code>text/html</code> .

Header	Type	Description
<code>Exchange.CONTENT_ENCODING</code>	String	The HTTP content encoding. Is set on both the IN and OUT message to provide a content encoding, such as <code>gzip</code> .

3.2.3. 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.

3.2.4. 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 `AhcOperationFailedException` with the information.



The option, `throwExceptionOnFailure`, can be set to `false` to prevent the `AhcOperationFailedException` from being thrown for failed response codes. This allows you to get any response from the remote server.

3.2.5. AhcOperationFailedException

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

3.2.6. 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.

3.2.7. Configuring URI to call

You can set the HTTP producer's URI directly from 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, `Exchange.HTTP_URI`, on the message.

```
from("direct:start")
  .setHeader(Exchange.HTTP_URI, constant("http://newhost"))
  .to("http4://oldhost");
```

3.2.8. Configuring URI Parameters

The **ahc** 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("ahc:http://oldhost?order=123&detail=short");
```

Or options provided in a header:

```
from("direct:start")
  .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
  .to("ahc://oldhost");
```

3.2.9. How to set the http method (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE) to the HTTP producer

The HTTP 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");
```

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

```
<to uri="http4://www.google.com"/>
<to uri="mock:results"/>
</route>
</camelContext>
```

3.2.10. 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");
```

3.2.10.1. 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("ahc:http://www.google.com/search?q=Camel", null);
```

3.2.10.2. 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("ahc:http://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.

3.2.10.3. Getting the Response Code

You can get the HTTP response code from the AHC component by getting the value from the Out message header with `Exchange.HTTP_RESPONSE_CODE`.

```
Exchange exchange =
    template.send("ahc:http://www.google.com/search", new Processor() {
        public void process(Exchange exchange) throws Exception {
            exchange.getIn().setHeader(Exchange.HTTP_QUERY,
                constant("hl=en&q=activemq"));
        }
    });
Message out = exchange.getOut();
int responseCode = out.getHeader(Exchange.HTTP_RESPONSE_CODE, Integer.class);
```

3.2.11. Configuring AsyncHttpClient

The `AsyncHttpClient` client uses a `AsyncHttpClientConfig` to configure the client. See the documentation at [Async Http Client](#) for more details.

In Camel 2.8, configuration is limited to using the builder pattern provided by `AsyncHttpClientConfig.Builder`. In Camel 2.8, the `AsyncHttpClientConfig` doesn't support getters/setters so its not easy to create/configure using a Spring bean style (eg the `<bean>` tag in the XML file).

The example below shows how to use a builder to create the `AsyncHttpClientConfig` which we configure on the `AhcComponent`.

```
// create a client config builder
AsyncHttpClientConfig.Builder builder = new AsyncHttpClientConfig.Builder();
// use the builder to set the options we want, in this case we want to follow
// redirects and try
// at most 3 retries to send a request to the host
AsyncHttpClientConfig config =
builder.setFollowRedirects(true).setMaxRequestRetry(3).build();

// lookup AhcComponent
AhcComponent component = context.getComponent("ahc", AhcComponent.class);
// and set our custom client config to be used
component.setClientConfig(config);
```

In Camel 2.9, the AHC component uses Async HTTP library 1.6.4. This newer version provides added support for plain bean style configuration. The `AsyncHttpClientConfigBean` class provides getters and setters for the configuration options available in `AsyncHttpClientConfig`. An instance of `AsyncHttpClientConfigBean` may be passed directly to the AHC component or referenced in an endpoint URI using the `clientConfig` URI parameter.

Also available in Camel 2.9 is the ability to set configuration options directly in the URI. URI parameters starting with "clientConfig." can be used to set the various configurable properties of `AsyncHttpClientConfig`. The properties specified in the endpoint URI are merged with those specified in the configuration referenced by the "clientConfig" URI parameter with those being set using the "clientConfig." parameter taking priority. The `AsyncHttpClientConfig` instance referenced is always copied for each endpoint such that settings on one endpoint will remain independent of settings on any previously created endpoints. The example below shows how to configure the AHC component using the "clientConfig." type URI parameters.

```
from("direct:start")
.to("ahc:http://localhost:8080/foo?clientConfig.maxRequestRetry=3&clientConfig.
followRedirects=true")
```

3.2.12. SSL Support (HTTPS)

The AHC 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 AHC 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);

AhcComponent component = context.getComponent("ahc", AhcComponent.class);
component.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="ahc:https://localhost/foo?sslContextParameters=#sslContextParameters"/>
...

```

3.3. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>

```

See the [Apache Camel website](#) for examples of this component in use.

3.3.1. URI format and options

```
atom://atomUri[?options]
```

Where **atomUri** is the URI to the Atom feed to poll.

And *options* can be:

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 <code>UpdateDateFilter</code> that only returns new entries from the feed. So the client consuming from the feed 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 <code>entry.updated</code> timestamp). Syntax format is: <code>yyyy-MM-ddTHH:MM:ss</code> . Example: <code>2007-12-24T17:45:59</code> .

Property	Default	Description
<code>throttleEntries</code>	<code>true</code>	Sets whether all entries identified in a single feed poll should be delivered immediately. If <code>true</code> , only one entry is processed per <code>consumer.delay</code> . Only applicable when <code>splitEntries</code> is set to <code>true</code> .
<code>feedHeader</code>	<code>true</code>	Sets whether to add the Abdera Feed object as a header.
<code>sortEntries</code>	<code>false</code>	If <code>splitEntries</code> is <code>true</code> , this sets whether to sort those entries by updated date.
<code>consumer.delay</code>	60000	Delay in milliseconds between each poll.
<code>consumer.initialDelay</code>	1000	Millis before polling starts.
<code>consumer.userFixedDelay</code>	<code>false</code>	If <code>true</code> , 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&...`

3.3.2. 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
<code>splitEntries</code>	<code>true</code>	Only a single entry from the currently being processed feed is set: <code>exchange.in.body(Entry)</code>
<code>splitEntries</code>	<code>false</code>	The entire list of entries from the feed is set: <code>exchange.in.body(List<Entry>)</code>

Camel can set the `Feed` object on the In header (see `feedHeader` option to disable this).

3.3.3. Message Headers

Camel atom uses these headers:

Header	Description
<code>CamelAtomFeed</code>	When consuming the <code>org.apache.abdera.model.Feed</code> object is set to this header.

3.4. Apns

The `apns` component is used for sending notifications to iOS devices. The `apns` components use [javapns](#) library.

The component supports sending notifications to Apple Push Notification Servers (APNS) and consuming feedback from the servers.

The consumer is configured with 3600 seconds for polling by default because it is a best practice to consume feedback stream from Apple Push Notification Servers only from time to time. For example: every 1 hour to avoid flooding the servers.

The feedback stream gives informations about inactive devices. You only need to get this informations every some hours if your mobile application is not a heavily used one.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-apns</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core
       version -->
</dependency>
```

3.4.1. URI format

To send notifications:

```
bean:beanID[?options]
```

To consume feedback:

```
bean:beanID[?options]
```

3.4.2. Options

Table 3.3. Producer

Option	Default	Description
tokens		Empty by default. Configure this property in case you want to statically declare tokens related to devices you want to notify. Tokens are separated by comma.

Table 3.4. Consumer

Option	Default	Description
delay	3600	Delay in seconds between each poll.
initialDelay	10	Seconds before polling starts.
timeUnit	SECONDS	Time Unit for polling.
userFixedDelay	true	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&...`

3.4.3. Component

The `ApnsComponent` must be configured with a `com.notnoop.apns.ApnsService`. The service can be created and configured using the `org.apache.camel.component.apns.factory.ApnsServiceFactory`. See further below for an example. And as well in the [test source code](#).

3.4.4. Exchange data format

When Camel will fetch feedback data corresponding to inactive devices, it will retrieve a List of `InactiveDevice` objects. Each `InactiveDevice` object of the retrieved list will be setted as the In body, and then processed by the consumer endpoint.

3.4.5. Message Headers

Camel Apns uses these headers.

Property	Default	Description
CamelApnsTokens		Empty by default.
CamelApnsMessageType	STRING, PAYLOAD	In case you choose PAYLOAD for the message type, then the message will be considered as a APNS payload and sent as is. In case you choose STRING, message will be converted as a APNS payload

3.4.6. Samples

3.4.6.1. Camel Xml route

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:camel="http://camel.apache.org/schema/spring"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://camel.apache.org/schema/spring
    http://camel.apache.org/schema/spring/camel-spring.xsd">

  <!-- Replace by desired values -->
  <bean id="apnsServiceFactory"
    class="org.apache.camel.component.apns.factory.ApnsServiceFactory">

    <!-- Optional configuration of feedback host and port -->
    <!-- <property name="feedbackHost" value="localhost" /> -->
    <!-- <property name="feedbackPort" value="7843" /> -->

    <!-- Optional configuration of gateway host and port -->
    <!-- <property name="gatewayHost" value="localhost" /> -->
    <!-- <property name="gatewayPort" value="7654" /> -->

    <!-- Declaration of certificate used -->
    <!-- from Camel 2.11 onwards you can use prefix: classpath:,
      file: to refer to load the certificate from classpath or file.
      Default it classpath -->
    <property name="certificatePath" value="certificate.pl2" />
    <property name="certificatePassword" value="MyCertPassword" />

    <!-- Optional connection strategy - By Default: No need to configure -->
    <!-- Possible options: NON_BLOCKING, QUEUE, POOL or Nothing -->
    <!-- <property name="connectionStrategy" value="POOL" /> -->
    <!-- Optional pool size -->
    <!-- <property name="poolSize" value="15" /> -->

    <!-- Optional connection strategy - By Default: No need to configure -->
    <!-- Possible options: EVERY_HALF_HOUR, EVERY_NOTIFICATION or Nothing
      (Corresponds to NEVER javapns option) -->
    <!-- <property name="reconnectionPolicy" value="EVERY_HALF_HOUR" /> -->
  </bean>

  <bean id="apnsService" factory-bean="apnsServiceFactory"
    factory-method="getApnsService" />

  <!-- Replace this declaration by wanted configuration -->
  <bean id="apns" class="org.apache.camel.component.apns.ApnsComponent">
```



```

        <property name="apnsService" ref="apnsService" />
    </bean>

    <camelContext id="camel-apns-test"
        xmlns="http://camel.apache.org/schema/spring">
        <route id="apns-test">
            <from
                uri="apns:consumer?initialDelay=10&delay=3600&timeUnit=SECONDS" />
            <to
                uri="log:org.apache.camel.component.apns?showAll=true&multiline=true" />
            <to uri="mock:result" />
        </route>
    </camelContext>
</beans>

```

3.4.6.2. Camel Java route

Create camel context and declare apns component programmatically:

```

protected CamelContext createCamelContext() throws Exception {
    CamelContext camelContext = super.createCamelContext();

    ApnsServiceFactory apnsServiceFactory = new ApnsServiceFactory();
    apnsServiceFactory.setCertificatePath("classpath:/certificate.pl2");
    apnsServiceFactory.setCertificatePassword("MyCertPassword");

    ApnsService apnsService = apnsServiceFactory.getApnsService(camelContext);

    ApnsComponent apnsComponent = new ApnsComponent(apnsService);
    camelContext.addComponent("apns", apnsComponent);

    return camelContext;
}

```

ApnsProducer - iOS target device dynamically configured via header: "CamelApnsTokens"

```

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:test")
                .setHeader(ApnsConstants.HEADER_TOKENS, constant(IOS_DEVICE_TOKEN))
                .to("apns:notify");
        }
    }
}

```

ApnsProducer - iOS target device statically configured via uri:

```

protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:test").
                to("apns:notify?tokens=" + IOS_DEVICE_TOKEN);
        }
    };
}

```

ApnsConsumer:

```

from("apns:consumer?initialDelay=10&delay=3600&timeUnit=SECONDS")
    .to("log:com.apache.camel.component.apns?showAll=true&multiline=true")

```

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

3.5. Avro

This component provides a dataformat for avro, which allows serialization and deserialization of messages using Apache Avro's binary dataformat. Moreover, it provides support for Apache Avro's rpc, by providing producers and consumers endpoint for using avro over netty or 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-avro</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.5.1. Apache Avro Overview

Avro allows you to define message types and a protocol using a json like format and then generate java code for the specified types and messages. An example of how a schema looks like is below.

```
{ "namespace": "org.apache.camel.avro.generated",
  "protocol": "KeyValueProtocol",

  "types": [
    { "name": "Key", "type": "record",
      "fields": [
        { "name": "key", "type": "string" }
      ]
    },
    { "name": "Value", "type": "record",
      "fields": [
        { "name": "value", "type": "string" }
      ]
    }
  ],

  "messages": {
    "put": {
      "request": [{ "name": "key", "type": "Key" }, { "name": "value", "type": "Value" } ],
      "response": "null"
    },
    "get": {
      "request": [{ "name": "key", "type": "Key" } ],
      "response": "Value"
    }
  }
}
```

You can easily generate classes from a schema, using maven, ant etc. More details can be found at the [Apache Avro documentation](#).

However, it doesn't enforce a schema first approach and you can create schema for your existing classes. Since 2.12 you can use existing protocol interfaces to make RCP calls. You should use interface for the protocol itself and POJO beans or primitive/String classes for parameter and result types. Here is an example of the class that corresponds to schema above:

```
package org.apache.camel.avro.reflection;

public interface KeyValueProtocol {
    void put(String key, Value value);
    Value get(String key);
}

class Value {
    private String value;
    public String getValue() { return value; }
    public void setValue(String value) { this.value = value; }
}
```

Existing classes can be used only for RPC (see below), not in data format.

3.5.2. Using the Avro data format

Using the avro data format is as easy as specifying that the class that you want to marshal or unmarshal in your route.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="direct:in"/>
    <marshal>
      <avro instanceClass="org.apache.camel.dataformat.avro.Message"/>
    </marshal>
    <to uri="log:out"/>
  </route>
</camelContext>
```

An alternative can be to specify the dataformat inside the context and reference it from your route.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
  <dataFormats>
    <avro id="avro" instanceClass="org.apache.camel.dataformat.avro.Message"/>
  </dataFormats>
  <route>
    <from uri="direct:in"/>
    <marshal ref="avro"/>
    <to uri="log:out"/>
  </route>
</camelContext>
```

In the same manner you can umarshal using the avro data format.

3.5.3. Using Avro RPC in Camel

As mentioned above Avro also provides RPC support over multiple transports such as http and netty. Camel provides consumers and producers for these two transports.

```
avro:[transport]:[host]:[port][?options]
```

The supported transport values are currently http or netty.

Since 2.12 you can specify message name right in the URI:

```
avro:[transport]:[host]:[port][/messageName][?options]
```

For consumers this allows you to have multiple routes attached to the same socket. Dispatching to correct route will be done by the avro component automatically. Route with no `messageName` specified (if any) will be used as default.

When using camel producers for avro ipc, the "in" message body needs to contain the parameters of the operation specified in the avro protocol. The response will be added in the body of the "out" message.

In a similar manner when using camel avro consumers for avro ipc, the requests parameters will be placed inside the "in" message body of the created exchange and once the exchange is processed the body of the "out" message will be send as a response.

By default consumer parameters are wrapped into array. If you've got only one parameter, since 2.12 you can use `singleParameter` URI option to receive it directly in the "in" message body without array wrapping.

3.5.4. Avro RPC URI Options

Name	Version	Description
<code>protocolClassName</code>		The class name of the avro protocol.
<code>singleParameter</code>	2.12	If true, consumer parameter won't be wrapped into array. Will fail if protocol specifies more then 1 parameter for the message.
<code>protocol</code>		Avro procol object. Can be used instead of <code>protocolClassName</code> when complex protocol needs to be created. One cane used <code>#name</code> notation to refer beans from the Registry
<code>reflectionProtocol</code>	2.12	If protocol object provided is reflection protocol. Should be used only with <code>protocol</code> parameter because for <code>protocolClassName</code> protocol type will be autodetected.

3.5.5. Avro RPC Headers

Name	Description
<code>CamelAvroMessageName</code>	The name of the message to send. In consumer overrides message name from URI (if any)

3.5.6. Examples

An example of using camel avro producers via http:

```
<route>
  <from uri="direct:start"/>
  <to uri="avro:http:localhost:{{avroport}}?protocolClassName=
    org.apache.camel.avro.generated.KeyValueProtocol"/>
  <to uri="log:avro"/>
</route>
```

In the example above you need to fill `CamelAvroMessageName` header. Since 2.12 you can use following syntax to call constant messages:

```
<route>
  <from uri="direct:start"/>
  <to uri="avro:http:localhost:{{avroport}}/put?protocolClassName=
    org.apache.camel.avro.generated.KeyValueProtocol"/>
  <to uri="log:avro"/>
</route>
```

```
</route>
```

An example of consuming messages using camel avro consumers via netty:

```
<route>
  <from uri="avro:netty:localhost:{avroport}?protocolClassName=
    org.apache.camel.avro.generated.KeyValueProtocol"/>
  <choice>
    <when>
      <el>${in.headers.CamelAvroMessageName == 'put'}</el>
      <process ref="putProcessor"/>
    </when>
    <when>
      <el>${in.headers.CamelAvroMessageName == 'get'}</el>
      <process ref="getProcessor"/>
    </when>
  </choice>
</route>
```

Since 2.12 you can set up two distinct routes to perform the same task:

```
<route>
  <from uri="avro:netty:localhost:{avroport}"/put?protocolClassName=
    org.apache.camel.avro.generated.KeyValueProtocol">
  <process ref="putProcessor"/></route>
<route>
  <from uri="avro:netty:localhost:{avroport}"/get?protocolClassName=
    org.apache.camel.avro.generated.KeyValueProtocol&singleParameter=true"/>
  <process ref="getProcessor"/>
</route>
```

In the example above, `get` takes only one parameter, so `singleParameter` is used and `getProcessor` will receive `Value` class directly in body, while `putProcessor` will receive an array of size 2 with `String` key and `Value` value filled as array contents.

3.6. Bean

The **bean:** component binds beans to Camel message exchanges.

3.6.1. URI format and options

```
bean:beanID[?options]
```

Where **beanID** can be any string which is used to look up the bean in the Camel Registry.

And *options* can be:

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. You can specify type qualifiers to pin-point the exact method to use for overloaded methods, as well as specify parameter values directly in the method syntax. See more details at "Bean Binding" below.
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.

Name	Type	Default	Description
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&...`

3.6.2. Usage

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 a 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>
```

3.6.3. Bean as endpoint

Camel also supports invoking [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.

3.6.4. Java DSL bean syntax

Java DSL comes with syntactic sugar for the [Bean] component. Instead of specifying the bean explicitly as the endpoint (i.e. `to("bean:beanName")`) you can use the following syntax:

```
// Send message to the bean endpoint
// and invoke method resolved using Bean Binding.
from("direct:start").beanRef("beanName");

// Send message to the bean endpoint
// and invoke given method.
from("direct:start").beanRef("beanName", "methodName");
```

Instead of passing name of the reference to the bean (so that Camel will lookup for it in the registry), you can specify the bean itself:

```
// Send message to the given bean instance.
from("direct:start").bean(new ExampleBean());

// Explicit selection of bean method to be invoked.
from("direct:start").bean(new ExampleBean(), "methodName");

// Camel will create the instance of bean and cache it for you.
from("direct:start").bean(ExampleBean.class);
```

3.6.5. 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.

3.7. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.7.1. URI format and Options

```
cache://cacheName[?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`, where *option* can be:

Name	Default Value	Description
<code>maxElementsInMemory</code>	1000	The number of elements that may be stored in the defined cache
<code>memoryStore-EvictionPolicy</code>	<code>MemoryStore-Eviction-Policy.LFU</code>	The number of elements that may be stored in the defined cache. Options include <ul style="list-style-type: none"> <code>MemoryStoreEvictionPolicy.LFU</code> - Least frequently used <code>MemoryStoreEvictionPolicy.LRU</code> - Least recently used <code>MemoryStoreEvictionPolicy.FIFO</code> - first in first out, the oldest element by creation time
<code>overflowToDisk</code>	<code>true</code>	Specifies whether cache may overflow to disk
<code>eternal</code>	<code>false</code>	Sets whether elements are eternal. If eternal, timeouts are ignored and the element never expires.
<code>timeToLiveSeconds</code>	300	The maximum time between creation time and when an element expires. Is used only if the element is not eternal.
<code>timeToIdleSeconds</code>	300	The maximum amount of time between accesses before an element expires
<code>diskPersistent</code>	<code>false</code>	Whether the disk store persists between restarts of the Virtual Machine.
<code>diskExpiryThread-IntervalSeconds</code>	120	The number of seconds between runs of the disk expiry thread.
<code>cacheManagerFactory</code>	<code>null</code>	If you want to use a custom factory which instantiates and creates the <code>EHCACHE net.sf.ehcache.CacheManager</code> . Use type of <code>abstract org.apache.camel.component.cache.CacheManagerFactory</code> .
<code>eventListenerRegistry</code>	<code>null</code>	Sets a list of <code>EHCACHE net.sf.ehcache.event.CacheEventListener</code> for all new caches - no need to define it per cache in <code>EHCACHE xml config</code> anymore. Use type of <code>org.apache.camel.component.cache.CacheEventListenerRegistry</code> .
<code>cacheLoaderRegistry</code>	<code>null</code>	Sets a list of <code>org.apache.camel.component.cache.CacheLoaderWrapper</code> that extends <code>EHCACHE net.sf.ehcache.loader.CacheLoader</code> for all new caches - no need to define it per cache in <code>EHCACHE xml config</code> anymore. Use type of <code>org.apache.camel.component.cache.CacheLoaderRegistry</code>
<code>key</code>	<code>null</code>	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.

Name	Default Value	Description
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.

3.7.2. Cache Component options

Name	Default Value	Description
configuration		To use a custom <code>org.apache.camel.component.cache.CacheConfiguration</code> configuration.
cacheManagerFactory		To use a custom <code>org.apache.camel.component.cache.CacheManagerFactory</code> .
configurationFile		Camel 2.13/2.12.3: To configure the location of the ehcache.xml file to use, such as <code>classpath:com/foo/mycache.xml</code> to load from classpath. If no configuration is given, then the default settings from EHCACHE is used.

3.7.3. Sending/Receiving Messages to/from the cache

3.7.3.1. 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 <ul style="list-style-type: none"> • 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.

Starting with Camel 2.11, the CamelCacheAdd and CamelCacheUpdate operations support additional headers:

Header	Type	Description
CamelCacheTimeToLive	Integer	Time to live in seconds
CamelCacheTimeToIdle	Integer	Time to idle in seconds
CamelCacheEternal	Integer	Whether the content is eternal

The CamelCacheAdd and CamelCacheUpdate operations support additional headers:

Header	Type	Description
CamelCacheTimeToLive	Integer	Camel 2.11: Time to live in seconds.

Header	Type	Description
CamelCacheTimeToIdle	Integer	Camel 2.11: Time to idle in seconds.
CamelCacheEternal	Integer	Camel 2.11: Whether the content is eternal.

3.7.3.2. 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

3.7.3.3. 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.

3.7.3.4. 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

3.7.4. Cache Usage Samples

3.7.4.1. Example: Configuring the cache

```
from("cache://MyApplicationCache" +  
    "?maxElementsInMemory=1000" +  
    "&memoryStoreEvictionPolicy=" +  
    "MemoryStoreEvictionPolicy.LFU" +  
    "&overflowToDisk=true" +  
    "&eternal=true" +  
    "&timeToLiveSeconds=300" +
```

```
"&timeToIdleSeconds=true" +
"&diskPersistent=true" +
"&diskExpiryThreadIntervalSeconds=300")
```

3.7.4.2. Example: 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")
    }
};
```

3.7.4.3. Example: 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")
    }
};
```

3.7.4.4. Example: 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")
    }
};
```

3.7.4.5. Example: 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");
    }
};
```

3.7.4.6. Example: 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
            }
        })
    }
};
```

3.7.4.7. Example: 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");
    }
};
```

3.7.4.8. Example: 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");

```

3.7.4.9. Example: 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();

```

3.7.5. 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.

3.8. Class

3.8.1. Class Component

The **class** component binds beans to Camel message exchanges. It works in the same way as the [Bean](#) component but instead of looking up beans from a [Registry](#) it creates the bean based on the class name.

3.8.1.1. URI format

```
class:className[?options]
```

where **className** is the fully qualified class name to create and use as bean.

3.8.1.2. 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 <code>true</code> , 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&...`

3.8.1.3. Using Class

You simply use the **class** component just as the [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");
```

3.8.2. 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 details at the [Bean](#) component as the `class` component works in much the same way.

3.9. CMIS

The `cmis` component uses the [Apache Chemistry](#) client API and allows you to add/read nodes to/from a CMIS compliant content repositories.

3.9.1. URI Format

```
cmis://cmisServerUrl[?options]
```

You can append query options to the URI in the following format, `?options=value&option2=value&...`

3.9.2. URI Options

Table 3.5.

Name	Default Value	Context	Description
<code>queryMode</code>	<code>false</code>	Producer	If true, will execute the <code>cmis</code> query from the message body and return result, otherwise will create a node in the <code>cmis</code> repository
<code>query</code>	<code>String</code>	Consumer	The <code>cmis</code> query to execute against the repository. If not specified, the consumer will retrieve every node from the content repository by iterating the content tree recursively
<code>username</code>	<code>null</code>	Both	Username for the <code>cmis</code> repository
<code>password</code>	<code>null</code>	Both	Password for the <code>cmis</code> repository
<code>repositoryId</code>	<code>null</code>	Both	The Id of the repository to use. If not specified the first available repository is used

Name	Default Value	Context	Description
pageSize	100	Both	Number of nodes to retrieve per page
readCount	0	Both	Max number of nodes to read
readContent	false	Both	If set to true, the content of document node will be retrieved in addition to the properties

3.9.3. Usage

3.9.3.1. Message headers evaluated by the producer

Table 3.6.

Header	Default Value	Description
CamelCMISFolderPath	/	The current folder to use during the execution. If not specified will use the root folder
CamelCMISRetrieveContent	false	In <code>queryMode</code> this header will force the producer to retrieve the content of document nodes.
CamelCMISReadSize	0	Max number of nodes to read.
cmis:path	null	If <code>CamelCMISFolderPath</code> is not set, will try to find out the path of the node from this cmis property and it is name
cmis:name	null	If <code>CamelCMISFolderPath</code> is not set, will try to find out the path of the node from this cmis property and it is path
cmis:objectTypeId	null	The type of the node
cmis:contentStreamMimeType	null	The mimetype to set for a document

3.9.3.2. Message headers set during querying Producer operation

Table 3.7.

Header	Type	Description
CamelCMISResultCount	Integer	Number of nodes returned from the query.

The message body will contain a list of maps, where each entry in the map is cmis property and its value. If `CamelCMISRetrieveContent` header is set to true, one additional entry in the map with key `CamelCMISContent` will contain `InputStream` of the document type of nodes.

3.9.4. Dependencies

Maven users will need to add the following dependency to their `pom.xml`.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cmis</artifactId>
  <version>${camel-version}</version>
</dependency>
```

where `${camel-version}` must be replaced by the actual version of Camel (2.11 or higher).

3.10. Context

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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.10.1. 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 **sed: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&...`

3.10.2. Example

In this example we'll create a black box context, then we'll use it from another CamelContext.

3.10.2.1. 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 **sed**). 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>

```

3.10.2.2. 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>

```

3.10.2.3. 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**

3.11. CouchDB

The couchdb component allows you to treat [CouchDB](#) instances as a producer or consumer of messages. Using the lightweight LightCouch API, this camel component has the following features:

- As a consumer, monitors couch changesets for inserts, updates and deletes and publishes these as messages into camel routes.
- As a producer, can save or update documents into couch.
- Can support as many endpoints as required, eg for multiple databases across multiple instances.
- Ability to have events trigger for only deletes, only inserts/updates or all (default).
- Headers set for sequenceId, document revision, document id, and HTTP method type.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-couchdb</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.11.1. URI format

```
couchdb:http://hostname[:port]/database?[options]
```

Where `hostname` is the hostname of the running couchdb instance. Port is optional and if not specified then defaults to 5984.

3.11.2. Options

Table 3.8.

Property	Default	Description
deletes	true	document deletes are published as events
updates	true	document inserts/updates are published as events
heartbeat	30000	how often to send an empty message to keep socket alive in millis
createDatabase	true	create the database if it does not already exist
username	null	username in case of authenticated databases
password	null	password for authenticated databases

3.11.3. Headers

The following headers are set on exchanges during message transport.

Headers are set by the consumer once the message is received. The producer will also set the headers for downstream processors once the insert/update has taken place. Any headers set prior to the producer are ignored.

That means for example, if you set CouchDbId as a header, it will not be used as the id for insertion, the id of the document will still be used.

3.11.4. Message Body

The component will use the message body as the document to be inserted. If the body is an instance of String, then it will be marshalled into a GSON object before insert. This means that the string must be valid JSON or the insert / update will fail. If the body is an instance of a `com.google.gson.JsonElement` then it will be inserted as is. Otherwise the producer will throw an exception of unsupported body type.

3.11.5. Samples

For example if you wish to consume all inserts, updates and deletes from a CouchDB instance running locally, on port 9999 then you could use the following:

```
from( "couchdb:http://localhost:9999" ).process( someProcessor ) ;
```

If you were only interested in deletes, then you could use the following:

```
from( "couchdb:http://localhost:9999?updates=false" ).process( someProcessor ) ;
```

If you wanted to insert a message as a document, then the body of the exchange is used:

```
from( "someProducingEndpoint" ).process( someProcessor )  
    .to( "couchdb:http://localhost:9999" )
```

3.12. Crypto (Digital Signatures)

With 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>  
  <!-- use the same version as your Camel core version -->  
  <version>x.x.x</version>  
</dependency>
```

3.12.1. 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. The following are some excellent resources for explaining the mechanics of Cryptography, Message digests and Digital Signatures and how to leverage them with the JCE.

- Bruce Schneier's Applied Cryptography
- Beginning Cryptography with Java by David Hook
- The ever insightful Wikipedia [Digital_signatures](#)

3.12.2. 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.

3.12.3. Options

Name	Type	Default	Description
algorithm	String	SHA1WithDSA	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.

Name	Type	Default	Description
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.
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").

3.13. CXF



When using CXF as a consumer, the [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.

When using CXF in streaming modes (see DataFormat option), then also read about [Stream caching](#).

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>
```



To learn about CXF dependencies the [WHICH-JARS](#) text file can be viewed.

3.13.1. 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:

```
cxfr:bean:cxfrEndpoint?wsdlURL=wsdl/hello_world.wsdl&dataFormat=PAYLOAD
```

3.13.2. Options

Name	Required	Description
wsdlURL	No	The location of the WSDL. It is obtained from endpoint address by default. <i>Example:</i> 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. 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. The serviceClass for a CXF producer (that is, the to endpoint) should be a Java interface. 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. <i>Example:</i> org.apache.camel.Hello
serviceClassInstance	Yes	Use either serviceClass or serviceClassInstance. <i>Deprecated in 2.x.</i> In 1.6.x serviceClassInstance works like serviceClass=#beanName, which looks up a serviceObject instance from the registry. <i>Example:</i> serviceClassInstance= beanName
serviceName	No	The service name this service is implementing, it maps to the wsdl:service@name <i>*Required</i> for camel-cxf consumer since camel-2.2.0 or if more than one serviceName is present in WSDL. <i>Example:</i> {http://org.apache.camel}ServiceName
endpointName	No	The port name this service is implementing, it maps to the wsdl:port@name <i>*Required</i> for camel-cxf consumer since camel-2.2.0 or if more than one portName is present under serviceName. <i>Example:</i> {http://org.apache.camel}PortName
dataFormat	No	The data type messages supported by the CXF endpoint. <i>Default:</i> POJO <i>Example:</i> 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 <i>Default:</i> true <i>Example:</i> true,false
wrapped	No	Which kind of operation that CXF endpoint producer will invoke. <i>Default:</i> false

Name	Required	Description
		<i>Example:</i> true, false
wrappedStyle	No	<p>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.</p> <p><i>Default:</i> Null</p> <p><i>Example:</i> true, false</p>
setDefaultBus	No	<p>This will set the default bus when CXF endpoint create a bus by itself.</p> <p><i>Default:</i> false</p> <p><i>Example:</i> true, false</p>
bus	No	<p>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 <code>org.apache.cxf.Bus</code>.</p> <p><i>Example:</i> bus=#busName</p>
cxfBinding	No	<p>New in 2.0, use # notation to reference a CXF binding object from the registry. The referenced object must be an instance of <code>org.apache.camel.component.cxf.CxfBinding</code> (use an instance of <code>org.apache.camel.component.cxf.DefaultCxfBinding</code>).</p> <p><i>Example:</i> cxfBinding=#bindingName</p>
headerFilterStrategy	No	<p>Use # notation to reference a header filter strategy object from the registry. The referenced object must be an instance of <code>org.apache.camel.spi.HeaderFilterStrategy</code> (use an instance of <code>org.apache.camel.component.cxf.CxfHeaderFilterStrategy</code>).</p> <p><i>Example:</i> headerFilterStrategy=#strategyName</p>
loggingFeatureEnabled	No	<p>New in 2.3, this option enables CXF Logging Feature which writes inbound and outbound SOAP messages to log.</p> <p><i>Default:</i> false</p> <p><i>Example:</i> loggingFeatureEnabled=true</p>
defaultOperationName	No	<p>New in 2.4, this option will set the default operationName that will be used by the CxfProducer which invokes the remote service.</p> <p><i>Default:</i> null</p> <p><i>Example:</i> defaultOperationName=greetMe</p>
defaultOperationName-space	No	<p>New in 2.4, this option will set the default operationNamespace that will be used by the CxfProducer which invokes the remote service.</p> <p><i>Default:</i> null</p> <p><i>Example:</i> defaultOperationNamespace= http://apache.org/hello_world_soap_http</p>
synchronous	No	<p>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. <i>Default:</i> false</p> <p><i>Example:</i> synchronous=true</p>
publishedEndpointUrl	No	<p>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.</p> <p><i>Default:</i> null</p> <p><i>Example:</i> publishedEndpointUrl=http://example.com/service</p>
properties.XXX	No	<p>Allows for setting custom properties to CXF in the endpoint URI. For example setting <code>properties.mtom-enabled = true</code> to enable MTOM.</p>

Name	Required	Description
allowStreaming	No	This option controls whether the CXF component, when running in PAYLOAD mode, will parse the incoming messages into DOM Elements or keep the payload as a javax.xml.transform.Source object that would allow streaming in some cases.
skipFaultLogging	No	Starting in 2.11, this option controls whether the PhaseInterceptorChain skips logging Faults that it catches.
username	No	New in Camel 2.12.3 This option is used to set the basic authentication information of username for the CXF client.
password	No	New in Camel 2.12.3 This option is used to set the basic authentication information of password for the CXF client.
continuationTimeout	No	New in Camel 2.14.0 This option is used to set the CXF continuation timeout which could be used in CxfConsumer by default when the CXF server is using Jetty or Servlet transport. (Before Camel 2.14.0 , CxfConsumer just set the continuation timeout to be 0, which means the continuation suspend operation never timeout.) <i>Default:</i> 30000 <i>Example:</i> continuation=80000

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.

3.13.2.1. 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 <code>soap:body</code>) after message configuration in the CXF endpoint is applied. Only Protocol JAX-WS handlers are supported. Logical JAX-WS handlers are not.
MESSAGE	MESSAGE is the raw message that is received from the transport layer. It is not suppose to touch or change Stream, some of the CXF interceptors will be removed if you are using this kind of DataFormat so you can't see any soap headers after the camel-cxf consumer and JAX-WS handler is not supported.
CXF_MESSAGE	Starting with Camel 2.8.2, CXF_MESSAGE allows for invoking the full capabilities of CXF interceptors by converting the message from the transport layer into a raw SOAP message.

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`.

3.13.2.2. Logging Messages

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.

Note CXF's `LoggingOutInterceptor` outputs outbound messages that are sent on the wire to the 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>

```

3.13.2.3. 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
<code>relayHeaders</code>	All message headers will be processed by Message Header Filters	boolean	No	true
<code>relayAll-MessageHeaders</code>	All message headers will be propagated (without processing by Message Header Filters)	boolean	No	false
<code>allowFilter-NamespaceClash</code>	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

3.13.3. 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 <code>wsdl:port@name</code> . In the format of <code>ns:PORT_NAME</code> where <code>ns</code> is a namespace prefix valid at this scope.
serviceName	The service name this service is implementing, it maps to the <code>wsdl:service@name</code> . In the format of <code>ns:SERVICE_NAME</code> where <code>ns</code> 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 <code>bindingId</code> 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
<code>cxf:inInterceptors</code>	The incoming interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
<code>cxf:inFaultInterceptors</code>	The incoming fault interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
<code>cxf:outInterceptors</code>	The outgoing interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
<code>cxf:outFaultInterceptors</code>	The outgoing fault interceptors for this endpoint. A list of <code><bean></code> or <code><ref></code> .
<code>cxf:properties</code>	A properties map which should be supplied to the JAX-WS endpoint. See below.
<code>cxf:handlers</code>	A JAX-WS handler list which should be supplied to the JAX-WS endpoint. See below.
<code>cxf:dataBinding</code>	You can specify the which <code>DataBinding</code> will be use in the endpoint. This can be supplied using the Spring <code><bean class="MyDataBinding"/></code> syntax.
<code>cxf:binding</code>	You can specify the <code>BindingFactory</code> for this endpoint to use. This can be supplied using the Spring <code><bean class="MyBindingFactory"/></code> syntax.
<code>cxf:features</code>	The features that hold the interceptors for this endpoint. A list of <code>{{<bean>}}</code> s or <code>{{<ref>}}</code> s
<code>cxf:schemaLocations</code>	The schema locations for endpoint to use. A list of <code>{{<schemaLocation>}}</code> s
<code>cxf:serviceFactory</code>	The service factory for this endpoint to use. This can be supplied using the Spring <code><bean class="MyServiceFactory"/></code> syntax

You can find more advanced examples that show how to provide interceptors, properties and handlers on the CXF [JAX-WS Configuration page](#).

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>

```

3.13.4. 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});
    }
}

```

}

3.13.5. 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));
```

3.13.6. 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 docString = ECHO_RESPONSE;
    if (inElements.get(0).getLocalName().
        equals("echoBoolean")) {
        docString = 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(docString);
    outElements.add(outDocument.getDocumentElement());
    // set the payload header with null
    CxfPayload<SoapHeader> responsePayload =
        new CxfPayload<SoapHeader>(null, outElements);
    exchange.getOut().setBody(responsePayload);
}
}
}
}
}

```

3.13.7. 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);
}
}

```

3.13.8. 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());

```

3.13.9. SOAP headers are not available in MESSAGE mode

SOAP headers are not available in MESSAGE mode as SOAP processing is skipped.

3.13.10. 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);
```

3.13.11. 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 operation name",
    "{http://apache.org/hello_world_soap_http}greetMe",
    responseContext.get("javax.xml.ws.wsdl.operation").toString());

```

3.13.12. 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 2.12.3 . 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*.

```

<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 format to PAYLOAD mode -->
        <entry key="dataFormat" value="PAYLOAD"/>
    </cxf:properties>
</cxf:cxfEndpoint>

```

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.

```

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);

// skip "cid:"
String photoId = ele.getAttribute("href").substring(4);
Assert.assertEquals(MtomTestHelper.REQ_PHOTO_CID, photoId);

ele = (Element)xu.getValue("//ns:Detail/ns:image/xop:Include",
    in.getBody().get(0), XPathConstants.NODE);

// skip "cid:"
String imageId = ele.getAttribute("href").substring(4);
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.

3.14. CXF Bean Component

The **cxfbean:** component allows other Camel endpoints to send exchange and invoke Web service bean objects. **Currently, it only supports JAX-RS and JAX-WS (new to Camel 2.1) annotated service beans.**

Note : CxfBeanEndpoint is a ProcessorEndpoint so it has no consumers. It works similarly to a Bean component.

Maven users need to add the following dependency to their pom.xml to use the CXF Bean 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>

```

</dependency>

3.14.1. 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.

3.14.2. Options

Name	Required	Description
bus	No	<p>CXF bus reference specified by the # notation. The referenced object must be an instance of <code>org.apache.cxf.Bus</code>.</p> <p><i>Default:</i> Default bus created by CXF Bus Factory</p> <p><i>Example:</i> bus=#busName</p>
cxfBeanBinding	No	<p>CXF bean binding specified by the # notation. The referenced object must be an instance of <code>org.apache.camel.component.cxf.cxfbean.CxfBeanBinding</code>.</p> <p><i>Default:</i> DefaultCxfBeanBinding</p> <p><i>Example:</i> cxfBinding=#bindingName</p>
headerFilterStrategy	No	<p>Header filter strategy specified by the # notation. The referenced object must be an instance of <code>org.apache.camel.spi.HeaderFilterStrategy</code>.</p> <p><i>Default:</i> CxfHeaderFilterStrategy</p> <p><i>Example:</i> headerFilterStrategy=#strategyName</p>
populateFromClass	No	<p>Since 2.3, the <code>wsdlLocation</code> annotated in the POJO is ignored (by default) unless this option is set to <code>false</code>. Prior to 2.3, the <code>wsdlLocation</code> annotated in the POJO is always honored and it is not possible to ignore.</p> <p><i>Default:</i> true</p> <p><i>Example:</i> true,false</p>
providers	No	<p>Since 2.5, setting the providers for the CXFRS endpoint.</p> <p><i>Default:</i> null</p> <p><i>Example:</i> providers=#providerRef1,#providerRef2</p>
setDefaultBus	No	<p>This will set the default bus when CXF endpoint create a bus by itself.</p> <p><i>Default:</i> false</p> <p><i>Example:</i> true,false</p>

3.14.3. Headers



Currently, the CXF Bean component has (only) been tested with Jetty component -- it understands headers from Jetty component without requiring conversion.

Name	Required	Description
CamelHttp-CharacterEncoding	None	Character encoding

Name	Required	Description
		<i>Type:</i> String <i>In/Out:</i> In <i>Default:</i> None <i>Example:</i> ISO-8859-1
CamelContentType	No	Content type <i>Type:</i> String <i>Type:</i> String <i>Default:</i> */* <i>Example:</i> text/xml
CamelHttpBaseUri	Yes	The value of this header will be set in the CXF message as the <code>Message.BASE_PATH</code> property. It is needed by CXF JAX-RS processing. Basically, it is the scheme, host and port portion of the request URI. <i>Type:</i> String <i>In/Out:</i> In <i>Default:</i> The Endpoint URI of the source endpoint in the Camel exchange <i>Example:</i> http://localhost:9000
CamelHttpPath	Yes	Request URI's path <i>Type:</i> String <i>In/Out:</i> In <i>Default:</i> None <i>Example:</i> consumer/123
CamelHttpMethod	Yes	RESTful request verb <i>Type:</i> String <i>In/Out:</i> In <i>Default:</i> None <i>Example:</i> GET,PUT,POST,DELETE
CamelHttpResponseCode	No	HTTP response code <i>Type:</i> Integer <i>In/Out:</i> Out <i>Default:</i> None <i>Example:</i> 200

3.14.4. A Working Sample

This sample shows how to create a route that starts an embedded Jetty HTTP server. The route sends requests to a CXF Bean and invokes a JAX-RS 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 the RESTful request URI will not exactly match the endpoint's URI `http://localhost:9000`.

```
<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));
```

3.15. CXFRS



When using CXF as a consumer, the [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>
```

3.15.1. 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
```


3.15.2. Options

Name	Required	Description
resourceClasses	No	<p>The resource classes which you want to export as REST service. Multiple classes can be separated by comma.</p> <p><i>Default: None</i></p> <p><i>Example:</i> resourceClasses= org.apache.camel.rs.Example1, org.apache.camel.rs.Exchange2</p>
httpClientAPI	No	<p>new to Camel 2.1 If true, the CxfRsProducer will use the HttpClientAPI to invoke the service. If false, the CxfRsProducer will use the ProxyClientAPI to invoke the service.</p> <p><i>Default: true</i></p> <p><i>Example:</i> httpClientAPI=true</p>
synchronous	No	<p>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.</p> <p><i>Default: false</i></p> <p><i>Example:</i> synchronous=true</p>
throwExceptionOnFailure	No	<p>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.</p> <p><i>Default: true</i></p> <p><i>Example:</i> throwExceptionOnFailure=true</p>
maxClientCacheSize	No	<p>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.</p> <p><i>Default: 10</i></p> <p><i>Example:</i> maxClientCacheSize=5</p>
setDefaultBus	No	<p>New in 2.9.0. If true, will set the default bus when CXF endpoint create a bus by itself.</p> <p><i>Default: false</i></p> <p><i>Example:</i> setDefaultBus=true</p>
bus	No	<p>New in 2.9.0. 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.</p> <p><i>Default: None</i></p> <p><i>Example:</i> bus=#busName</p>
bindingStyle	No	<p>As of 2.11. Sets how requests and responses will be mapped to/from Camel. Two values are possible:</p> <ul style="list-style-type: none"> SimpleConsumer => see the Consuming a REST Request with the Simple Binding Style. Default => the default style. For consumers this passes on a MessageContentsList to the route, requiring low-level processing in the route. <p><i>Default: Default</i></p> <p><i>Example:</i> bindingStyle=SimpleConsumer</p>
providers	No	<p>Since Camel 2.12.2 set custom JAX-RS providers list to the CxfRs endpoint.</p>

Name	Required	Description
		<i>Default:</i> None <i>Example:</i> providers=#MyProviders
schemaLocations	No	Since Camel 2.12.2 Sets the locations of the schemas which can be used to validate the incoming XML or JAXB-driven JSON. <i>Default:</i> None <i>Example:</i> schemaLocations=#MySchemaLocations
features	No	Since Camel 2.12.3 Set the feature list to the CxfRs endpoint. <i>Default:</i> None <i>Example:</i> features=#MyFeatures
properties	No	Since Camel 2.12.4 Set the properties to the CxfRs endpoint. <i>Default:</i> None <i>Example:</i> properties=#MyProperties
inInterceptors	No	Since Camel 2.12.4 Set the inInterceptors to the CxfRs endpoint. <i>Default:</i> None <i>Example:</i> inInterceptors=#MyInterceptors
outInterceptors	No	Since Camel 2.12.4 Set the outInterceptor to the CxfRs endpoint. <i>Default:</i> None <i>Example:</i> outInterceptors=#MyInterceptors
inFaultInterceptors	No	Since Camel 2.12.4 Set the inFaultInterceptors to the CxfRs endpoint. <i>Default:</i> None <i>Example:</i> inFaultInterceptors=#MyInterceptors
outFaultInterceptors	No	Since Camel 2.12.4 Set the outFaultInterceptors to the CxfRs endpoint. <i>Default:</i> None <i>Example:</i> outFaultInterceptors=#MyInterceptors
continuationTimeout	No	Since Camel 2.14.0 This option is used to set the CXF continuation timeout which could be used in CxfConsumer by default when the CXF server is using Jetty or Servlet transport. (Before Camel 2.14.0 , CxfConsumer just set the continuation timeout to be 0, which means the continuation suspend operation never timeout.) <i>Default:</i> 30000 <i>Example:</i> continuationTimeout=800000

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 provide different configuration for them. Please check out the [schema file](#) and [CXF JAX-RS documentation](#) for more information.

See the [Camel Website](#) for the latest examples of this component in use.

3.16. 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.



The [SEDA](#) component provides asynchronous invocation of any consumers when a producer sends a message exchange.



The [VM](#) component provides connections between Camel contexts as long they run in the same **JVM**.

3.16.1. URI format

```
direct:someName[?options]
```

where **someName** can be any string to uniquely identify the endpoint.

3.16.2. Options

Name	Default Value	Description
allowMultipleConsumers	true	@deprecated If set to false, then when a second consumer is started on the endpoint, an <code>IllegalStateException</code> is thrown. Will be removed in Camel 2.1: Direct endpoint does not support multiple consumers.
block	false	Camel 2.11.1: If sending a message to a direct endpoint which has no active consumer, then we can tell the producer to block and wait for the consumer to become active.
timeout	30000	Camel 2.11.1: The timeout value to use if block is enabled.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.16.3. 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 [SEDA](#) component, how they can be used together.

3.17. Disruptor

The disruptor component provides asynchronous [SEDA](#) behavior much as the standard SEDA Component, but utilizes a [Disruptor](#) instead of a [BlockingQueue](#) utilized by the standard SEDA. Alternatively, a `disruptor-vm` endpoint is supported by this component, providing an alternative to the standard VM. As with the SEDA component, buffers of the disruptor endpoints are only visible within a single [CamelContext](#) and no support is provided for persistence or recovery. The buffers of the `*disruptor-vm:*` endpoints also provides support for communication across `CamelContexts` instances so you can use this mechanism to communicate across web applications (provided that `camel-disruptor.jar` is on the system/boot classpath).

The main advantage of choosing to use the Disruptor Component over the SEDA or the VM Component is performance in use cases where there is high contention between producer(s) and/or multicasted or concurrent Consumers. In those cases, significant increases of throughput and reduction of latency has been observed. Performance in scenarios without contention is comparable to the SEDA and VM Components.

The Disruptor is implemented with the intention of mimicing the behaviour and options of the SEDA and VM Components as much as possible. The main differences with the them are the following:

- The buffer used is always bounded in size (default 1024 exchanges).
- As a the buffer is always bouded, the default behaviour for the Disruptor is to block while the buffer is full instead of throwing an exception. This default behaviour may be configured on the component (see options).
- The Disruptor enpoints don't implement the `BrowsableEndpoint` interface. As such, the exchanges currently in the Disruptor can't be retrieved, only the amount of exchanges.
- The Disruptor requires its consumers (multicasted or otherwise) to be statically configured. Adding or removing consumers on the fly requires complete flushing of all pending exchanges in the Disruptor.
- As a result of the reconfiguration: Data sent over a Disruptor is directly processed and 'gone' if there is at least one consumer, late joiners only get new exchanges published after they've joined.
- The `pollTimeout` option is not supported by the Disruptor Component.
- When a producer blocks on a full Disruptor, it does not respond to thread interrupts.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-disruptor</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.17.1. URI format

```
disruptor:someName[?options]
```

or

```
disruptor-vm:someName[?options]
```

Where `*someName*` can be any string that uniquely identifies the endpoint within the current [CamelContext](#) (or across contexts in case of `*disruptor-vm:*`).

You can append query options to the URI in the following format:

?option=value&option=value&...

3.17.2. Options

All the following options are valid for both the **disruptor** and **disruptor-vm** components.

Name	Default	Description
size	1024	The maximum capacity of the Disruptors ringbuffer. Will be effectively increased to the nearest power of two. Notice: Mind if you use this option, then its the first endpoint being created with the queue name, that determines the size. To make sure all endpoints use same size, then configure the size option on all of them, or the first endpoint being created.
bufferSize		Component only: The maximum default size (capacity of the number of messages it can hold) of the Disruptors ringbuffer. This option is used if size is not in use.
queueSize		Component only: Additional option to specify the <code>bufferSize</code> to maintain maximum compatibility with the SEDA Component.
concurrentConsumers	1	Number of concurrent threads processing exchanges.
waitForTaskToComplete	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: <i>Always</i> , <i>Never</i> or <i>IfReplyExpected</i> . The first two values are self-explanatory. The last value, <i>IfReplyExpected</i> , will only wait if the message is Request Reply based. See more information about Async messaging.
timeout	30000	Timeout (in milliseconds) before a producer will stop waiting for an asynchronous task to complete. See <i>waitForTaskToComplete</i> and Async for more details. You can disable timeout by using 0 or a negative value.
defaultMultipleConsumers		Component only: Allows to set the default allowance of multiple consumers for endpoints created by this comonent used when <i>multipleConsumers</i> is not provided.
limitConcurrentConsumers	true	Whether to limit the number of concurrentConsumers to the maximum of 500. By default, an exception will be thrown if a Disruptor endpoint is configured with a greater number. You can disable that check by turning this option off.
blockWhenFull	true	Whether a thread that sends messages to a full Disruptor will block until the ringbuffer's capacity is no longer exhausted. By default, the calling thread will block and wait until the message can be accepted. By disabling this option, an exception will be thrown stating that the queue is full.
defaultBlockWhenFull		Component only: Allows to set the default allowance of multiple consumers for endpoints created by this comonent used when <i>multipleConsumers</i> is not provided.
waitStrategy	Blocking	Defines the strategy used by consumer threads to wait on new exchanges to be published. The options allowed are: <i>Blocking</i> , <i>Sleeping</i> , <i>BusySpin</i> and <i>Yielding</i> . Refer to the section below for more information on this subject.
defaultWaitStrategy		Component only: Allows to set the default wait strategy for endpoints created by this comonent used when <i>waitStrategy</i> is not provided.
producerType	Multi	Defines the producers allowed on the Disruptor. The options allowed are: <i>Multi</i> to allow multiple producers

Name	Default	Description
		and <i>Single</i> to enable certain optimizations only allowed when one concurrent producer (on one thread or otherwise synchronized) is active.
defaultProducerType		Component only: Allows to set the default producer type for endpoints created by this component used when <i>producerType</i> is not provided.

3.17.3. Wait strategies

The wait strategy effects the type of waiting performed by the consumer threads that are currently waiting for the next exchange to be published. The following strategies can be chosen:

Name	Description	Advice
Blocking	Blocking strategy that uses a lock and condition variable for Consumers waiting on a barrier.	This strategy can be used when throughput and low-latency are not as important as CPU resource.
Sleeping	Sleeping strategy that initially spins, then uses a Thread.yield(), and eventually for the minimum number of nanos the OS and JVM will allow while the Consumers are waiting on a barrier.	This strategy is a good compromise between performance and CPU resource. Latency spikes can occur after quiet periods.
BusySpin	Busy Spin strategy that uses a busy spin loop for Consumers waiting on a barrier.	Component only: Additional option to specify the <code>bufferSize</code> to maintain maximum compatibility with the SEDA Component.
Yielding	Yielding strategy that uses a Thread.yield() for Consumers waiting on a barrier after an initially spinning.	This strategy is a good compromise between performance and CPU resource without incurring significant latency spikes.

3.17.4. Use of Request Reply

The Disruptor component supports using [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("disruptor:input");
from("disruptor: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 *disruptor:input* buffer. As it is a [Request Reply](#) message, we wait for the response. When the consumer on the *disruptor:input* buffer is complete, it copies the response to the original message response.

3.17.5. Concurrent consumers

By default, the Disruptor 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("disruptor: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 and supported by the Disruptor internally so performance will be higher.

3.17.6. Thread pools

Be aware that adding a thread pool to a Disruptor endpoint by doing something like:

```
from("disruptor:stageName").thread(5).process(...)
```

Can wind up with adding a normal [BlockingQueue](#) to be used in conjunction with the Disruptor, effectively negating part of the performance gains achieved by using the Disruptor. Instead, it is advised to directly configure number of threads that process messages on a Disruptor endpoint using the `concurrentConsumers` option.

3.17.7. Sample

In the route below we use the Disruptor to send the request to this async queue to be able to send a fire-and-forget message for further processing in another thread, and return a constant reply in this thread to the original caller.

```
public void configure() throws Exception {
    from("direct:start")
        // send it to the disruptor that is async
        .to("disruptor:next")
        // return a constant response
        .transform(constant("OK"));

    from("disruptor:next").to("mock:result");
}
```

Here we send a Hello World message and expects the reply to be OK.

```
Object out = template.requestBody("direct:start", "Hello World");
assertEquals("OK", out);
```

The "Hello World" message will be consumed from the Disruptor from another thread for further processing. Since this is from a unit test, it will be sent to a mock endpoint where we can do assertions in the unit test.

3.17.8. Using multipleConsumers

In this example we have defined two consumers and registered them as spring beans.

```
<!-- define the consumers as spring beans -->
<bean id="consumer1" class="org.apache.camel.spring.example.FooEventConsumer"/>

<bean id="consumer2"
class="org.apache.camel.spring.example.AnotherFooEventConsumer"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- define a shared endpoint which the consumers can refer to instead of
    using url -->
    <endpoint id="foo" uri="disruptor:foo?multipleConsumers=true"/>
</camelContext>
```

Since we have specified `multipleConsumers=true` on the Disruptor foo endpoint we can have those two or more consumers receive their own copy of the message as a kind of pub-sub style messaging. As the beans are part of an unit test they simply send the message to a mock endpoint, but notice how we can use `@Consume` to consume from the Disruptor.

```
public class FooEventConsumer {
```

```

@EndpointInject(uri = "mock:result")
private ProducerTemplate destination;

@Consume(ref = "foo")
public void doSomething(String body) {
    destination.sendBody("foo" + body);
}
}

```

3.17.9. Extracting disruptor information

If needed, information such as buffer size, etc. can be obtained without using JMX in this fashion:

```

DisruptorEndpoint disruptor = context.getEndpoint("disruptor:xxxx");
int size = disruptor.getBufferSize();

```

3.18. ElasticSearch

The ElasticSearch component allows you to interface with an [ElasticSearch](#) server.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```

<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-elasticsearch</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>

```

3.18.1. URI format

```
elasticsearch://clusterName[?options]
```

if you want to run against a local (in JVM/classloader) ElasticSearch server, just set the `clusterName` value in the URI to "local". See the [client guide](#) for more details.

3.18.2. Endpoint Options

The following options may be configured on the ElasticSearch endpoint. All are required to be set as either an endpoint URI parameter or as a header (headers override endpoint properties)

name	description
operation	required, indicates the operation to perform
indexName	the name of the index to act against
indexType	the type of the index to act against
ip	the TransportClient remote host ip to use Camel 2.12
port	the TransportClient remote port to use (defaults to 9300) Camel 2.12

3.18.3. Message Operations

The following ElasticSearch operations are currently supported. Simply set an endpoint URI option or exchange header with a key of "operation" and a value set to one of the following. Some operations also require other parameters or the message body to be set.

operation	message body	description
INDEX	Map, String, byte[] or XContentBuilder content to index	adds content to an index and returns the content's indexId in the body
GET_BY_ID g	index id of content to retrieve	retrieves the specified index and returns a GetResult object in the body
DELETE	index id of content to delete	deletes the specified indexId and returns a DeleteResult object in the body

3.18.4. Index Example

Below is a simple INDEX example

```
from("direct:index")
  .to("elasticsearch://local?operation=INDEX&indexName=twitter&indexType=tweet");
```

```
<route>
  <from uri="direct:index" />
  <to
    uri="elasticsearch://local?operation=INDEX&indexName=twitter&indexType=tweet"/>
</route>
```

A client would simply need to pass a body message containing a Map to the route. The result body contains the indexId created.

```
Map<String, String> map = new HashMap<String, String>();
map.put("content", "test");
String indexId = template.requestBody("direct:index", map, String.class);
```

3.18.5. For more information, see these resources

[ElasticSearch Main Site](#)

[ElasticSearch Java API](#)

3.19. 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.

3.19.1. URI options

Name	Default value	Description
<code>args</code>	<code>null</code>	The arguments of the executable- they may be one or many whitespace-separated tokens, that can be quoted with <code>"</code> , for example, <code>args="arg 1" arg2</code> will use two arguments <code>arg 1</code> and <code>arg2</code> . To include the quotes, enclose them in another set of quotes; for example, <code>args="\"arg 1\"" arg2</code> will use the arguments <code>"arg 1"</code> and <code>arg2</code> .
<code>workingDir</code>	<code>null</code>	The directory in which the command should be executed. If <code>null</code> , the working directory of the current process will be used.
<code>timeout</code>	<code>Long.MAX_VALUE</code>	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.
<code>outFile</code>	<code>null</code>	The name of a file, created by the executable, that should be considered as output of the executable. If no <code>outFile</code> is set, the standard output (<code>stdout</code>) of the executable will be used instead.
<code>binding</code>	a <code>DefaultExecBinding</code> instance	A reference to an <code>org.apache.commons.exec.ExecBinding</code> in the Registry .
<code>commandExecutor</code>	a <code>DefaultCommand-Executor</code> instance	A reference to an <code>org.apache.commons.exec.ExecCommandExecutor</code> 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.
<code>useStderrOnEmpty-Stdout</code>	<code>false</code>	A boolean which dictates when <code>stdin</code> is empty, it should fallback and use <code>stderr</code> in the Camel Message Body. This option is default <code>false</code> .

3.19.2. Message headers

The supported headers are defined in `org.apache.camel.component.exec.ExecBinding`.

Name	Message	Description
<code>ExecBinding.EXEC_COMMAND_EXECUTABLE</code>	in	The name of the system command that will be executed. Overrides the <code>executable</code> in the URI. <i>Type:</i> <code>String</code>
<code>ExecBinding.EXEC_COMMAND_ARGS</code>	in	The arguments of the executable. The arguments are used literally, no quoting is applied. Overrides existing <code>args</code> in the URI. <i>Type:</i> <code>java.util.List<String></code>
<code>ExecBinding.EXEC_COMMAND_ARGS</code>	in	The arguments of the executable as a single string where each argument is whitespace separated (see <code>args</code> in URI option). The arguments are used literally, no quoting is applied. Overrides existing <code>args</code> in the URI.

Name	Message	Description
		<i>Type:</i> 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 <code>outFile</code> in the URI. <i>Type:</i> String
ExecBinding. EXEC_COMMAND_TIMEOUT	in	The timeout, in milliseconds, after which the executable should be terminated. Overrides existing <code>timeout</code> in the URI. <i>Type:</i> long
ExecBinding. EXEC_COMMAND_WORKING_DIR	in	The directory in which the command should be executed. Overrides existing <code>workingDir</code> in the URI. <i>Type:</i> String
ExecBinding.EXEC_EXIT_VALUE	out	The value of this header is the <i>exit value</i> of the executable. Non-zero exit values typically indicate abnormal termination. Note that the exit value is OS-dependent. <i>Type:</i> int
ExecBinding.EXEC_STDERR	out	The value of this header points to the standard error stream (<code>stderr</code>) of the executable. If no <code>stderr</code> is written, the value is null. <i>Type:</i> java.io.InputStream
ExecBinding. EXEC_USE_STDERR_ON_EMPTY_STDOUT	in	Indicates when the <code>stdin</code> is empty, should we fallback and use <code>stderr</code> as the body of the Camel message. By default this option is false. <i>Type:</i> boolean

3.19.3. 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
        assertInstanceOf(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
...
}
});
```

3.20. Facebook

Available as of Camel 2.12

The Facebook component provides access to all of the Facebook APIs accessible using [Facebook4J](https://developers.facebook.com/docs/facebook-sdk). It allows producing messages to retrieve, add, and delete posts, likes, comments, photos, albums, videos, photos, checkins, locations, links, etc. It also supports APIs that allow polling for posts, users, checkins, groups, locations, etc.

Facebook requires the use of OAuth for all client application authentication. In order to use camel-facebook with your account, you'll need to create a new application within Facebook at <https://developers.facebook.com/apps> and grant the application access to your account. The Facebook application's id and secret will allow access to Facebook APIs which do not require a current user. A user access token is required for APIs that require a logged in user. More information on obtaining a user access token can be found at <https://developers.facebook.com/docs/facebook-login/access-tokens/>.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-facebook</artifactId>
  <version>${camel-version}</version>
</dependency>
```

3.20.1. URI format

```
facebook://[endpoint]?[options]
```

3.20.2. FacebookComponent

The facebook component can be configured with the Facebook account settings below, which are mandatory. The values can be provided to the component using the bean property **configuration** of type **org.apache.camel.component.facebook.config.FacebookConfiguration**. The **oAuthAccessToken** option may be omitted but that will only allow access to application APIs.

You can also configure these options directly in an endpoint URI.

Option	Description
oAuthAppId	The application Id
oAuthAppSecret	The application Secret
oAuthAccessToken	The user access token

In addition to the above settings, non-mandatory options below can be used to configure the underlying Facebook4J runtime through either the component's **configuration** property or in an endpoint URI.

Option	Description	Default Value
oAuthAuthorizationURL	OAuth authorization URL	https://www.facebook.com/dialog/oauth
oAuthPermissions	Default OAuth permissions. Comma separated permission names. See https://developers.facebook.com/docs/reference/login/#permissions for the detail	null
oAuthAccessTokenURL	OAuth access token URL	https://developers.facebook.com/docs/facebook-login/access-tokens
debugEnabled	Enables debug output. Effective only with the embedded logger	false
gzipEnabled	Use Facebook GZIP encoding	true
httpConnectionTimeout	Http connection timeout in milliseconds	20000
httpDefaultMaxPerRoute	HTTP maximum connections per route	2
httpMaxTotalConnections	HTTP maximum total connections	20
httpProxyHost	HTTP proxy server host name	null
httpProxyPassword	HTTP proxy server password	null
httpProxyPort	HTTP proxy server port	null
httpProxyUser	HTTP proxy server user name	null
httpReadTimeout	Http read timeout in milliseconds	120000
httpRetryCount	Number of HTTP retries	0
httpRetryIntervalSeconds	HTTP retry interval in seconds	5
httpStreamingReadTimeout	HTTP streaming read timeout in milliseconds	40000
jsonStoreEnabled	If set to true, raw JSON forms will be stored in DataObjectFactory	false
mbeanEnabled	If set to true, Facebook4J mbean will be registered	false
prettyDebugEnabled	pretty JSON debug output if set to true	false
restBaseURL	API base URL	https://graph.facebook.com/
useSSL	Use SSL	true
videoBaseURL	Video API base URL	https://graph-video.facebook.com/
clientURL	Facebook4J API client URL	<a href="http://facebook4j.org/en/facebook4j-<version>.xml">http://facebook4j.org/en/facebook4j-<version>.xml
clientVersion	Facebook4J client API version	1.1.12

3.20.3. Producer Endpoints:

Producer endpoints can use endpoint names and options from the table below. Endpoints can also use the short name without the **get** or **search** prefix, except **checkin** due to ambiguity between **getCheckin** and **searchCheckin**. Endpoint options that are not mandatory are denoted by [].

Producer endpoints can also use a special option ***inBody*** that in turn should contain the name of the endpoint option whose value will be contained in the Camel Exchange In message. For example, the facebook endpoint in the following route retrieves activities for the user id value in the incoming message body.

```
from("direct:test").to("facebook://activities?inBody=userId")...
```

Any of the endpoint options can be provided in either the endpoint URI, or dynamically in a message header. The message header name must be of the format **CamelFacebook.option**. For example, the **userId** option value in the previous route could alternately be provided in the message header **CamelFacebook.userId**. Note that the **inBody** option overrides message header, e.g. the endpoint option **inBody=user** would override a **CamelFacebook.userId** header.

Endpoints that return a String return an Id for the created or modified entity, e.g. **addAlbumPhoto** returns the new album Id. Endpoints that return a boolean, return true for success and false otherwise. In case of Facebook API errors the endpoint will throw a `RuntimeException` with a `facebook4j.FacebookException` cause.

Endpoint	Short Name	Options	Body Type
Accounts			
getAccounts	accounts	[reading],[userId]	facebook4j.ResponseList <facebook4j.Account>
Activities			
getActivities	activities	[reading],[userId]	facebook4j.ResponseList <facebook4j.Activity>
Albums			
addAlbumPhoto	addAlbumPhoto	albumId,source, [message]	String
commentAlbum	commentAlbum	albumId,message	String
createAlbum	createAlbum	albumCreate, [userId]	String
getAlbum	album	albumId,[reading]	facebook.Album
getAlbumComments	albumComments	albumId,[reading]	facebook4j.ResponseList <facebook4j.Comment>
getAlbumCoverPhoto	albumCoverPhoto	albumId	java.net.URL
getAlbumLikes	albumLikes	albumId,[reading]	facebook4j.ResponseList <facebook4j.Like>
getAlbumPhotos	albumPhotos	albumId,[reading]	facebook4j.ResponseList <facebook4j.Photos>
getAlbums	albums	[reading],[userId]	facebook4j.ResponseList <facebook4j.Album>
likeAlbum	likeAlbum	albumId	boolean
unlikeAlbum	unlikeAlbum	albumId	boolean
Checkins			
checkin	checkin	checkinCreate, [userId]	String
commentCheckin	commentCheckin	checkinId, message	String
getCheckin	checkin	checkinId, [reading]	facebook4j.Checkin
getCheckinComments	checkinComments	checkinId, [reading]	facebook4j.ResponseList <facebook4j.Comment>
getCheckinLikes	checkinLikes	checkinId, [reading]	facebook4j.ResponseList <facebook4j.Like>
getCheckins	checkins	[reading], [userId]	facebook4j.ResponseList <facebook4j.Checkin>
likeCheckin	likeCheckin	checkinId	boolean
unlikeCheckin	unlikeCheckin	checkinId	boolean
Comments			
deleteComment	deleteComment	commentId	boolean
getComment	comment	commentId	facebook4j.Comment
getCommentLikes	commentLikes	commentId, [reading]	facebook4j.ResponseList

Endpoint	Short Name	Options	Body Type
			<facebook4j.Like>
likeComment	likeComment	commentId	boolean
unlikeComment	unlikeComment	commentId	boolean
Domains			
getDomain	domain	domainId	facebook4j.Domain
getDomainByName	domainByName	domainName	facebook4j.Domain
getDomainsByName	domainsByName	domainNames	java.util.List <facebook4j.Domain>
Events			
createEvent	createEvent	eventUpdate, [userId]	String
deleteEvent	deleteEvent	eventId	boolean
deleteEventPicture	deleteEventPicture	eventId	boolean
editEvent	editEvent	eventId, eventUpdate	boolean
getEvent	event	eventId,[reading]	facebook4j.Event
getEventFeed	eventFeed	eventId,[reading]	facebook4j.ResponseList <facebook4j.Post>
getEventPhotos	eventPhotos	eventId,[reading]	facebook4j.ResponseList <facebook4j.Photo>
getEventPictureURL	eventPictureURL	eventId,[size]	java.net.URL
getEvents	events	[reading],[userId]	facebook4j.ResponseList <facebook4j.Event>
getEventVideos	eventVideos	eventId,[reading]	facebook4j.ResponseList <facebook4j.Video>
getRSVPStatusAsInvited	rsvpStatusAsInvited	eventId,[userId]	facebook4j.ResponseList <facebook4j.RSVPStatus>
getRSVPStatusAsNoreply	rsvpStatusAsNoreply	eventId,[userId]	facebook4j.ResponseList <facebook4j.RSVPStatus>
getRSVPStatusInAttending	rsvpStatusInAttending	eventId,[userId]	facebook4j.ResponseList <facebook4j.RSVPStatus>
getRSVPStatusInDeclined	rsvpStatusInDeclined	eventId,[userId]	facebook4j.ResponseList <facebook4j.RSVPStatus>
getRSVPStatusInMaybe	rsvpStatusInMaybe	eventId,[userId]	facebook4j.ResponseList <facebook4j.RSVPStatus>
inviteToEvent	inviteToEvent	eventId,[userId], [userIds]	boolean
postEventFeed	postEventFeed	eventId, postUpdate	String
postEventLink	postEventLink	eventId,link, [message]	String
postEventPhoto	postEventPhoto	eventId,source, [message]	String
postEventStatusMessage	postEventStatusMessage	eventId,message	String
postEventVideo	postEventVideo	eventId,source, [title,description]	String

Endpoint	Short Name	Options	Body Type
rsvpEventAsAttending	rsvpEventAsAttending	eventId	boolean
rsvpEventAsDeclined	rsvpEventAsDeclined	eventId	boolean
rsvpEventAsMaybe	rsvpEventAsMaybe	eventId	boolean
uninviteFromEvent	uninviteFromEvent	eventId,userId	boolean
updateEventPicture	updateEventPicture	eventId,source	boolean
Family			
getFamily	family	[reading],[userId]	facebook4j.ResponseList <facebook4j.Family>
Favorites			
getBooks	books	[reading],[userId]	facebook4j.ResponseList <facebook4j.Book>
getGames	games	[reading],[userId]	facebook4j.ResponseList <facebook4j.Game>
getInterests	interests	[reading],[userId]	facebook4j.ResponseList <facebook4j.Interest>
getMovies	movies	[reading],[userId]	facebook4j.ResponseList <facebook4j.Movie>
getMusic	music	[reading],[userId]	facebook4j.ResponseList <facebook4j.Music>
getTelevision	television	[reading],[userId]	facebook4j.ResponseList <facebook4j.Television>
Facebook Query Language (FQL)			
executeFQL	executeFQL	query,[locale]	facebook4j.internal.org. json.JSONArray
executeMultiFQL	executeMultiFQL	queries,[locale]	java.util.Map<String,facebook4j. internal.org.json.JSONArray>
Friends			
addFriendlistMember	addFriendlistMember	friendlistId, userId	boolean
createFriendlist	createFriendlist	friendlistName, [userId]	String
deleteFriendlist	deleteFriendlist	friendlistId	boolean
getBelongsFriend	belongsFriend	friendId,[reading], [userId]	facebook4j.ResponseList <facebook4j.Friend>
getFriendlist	friendlist	friendlistId, [reading]	facebook4j.FriendList
getFriendlistMembers	friendlistMembers	friendlistId	facebook4j.ResponseList <facebook4j.Friend>
getFriendlists	friendlists	[reading],[userId]	facebook4j.ResponseList <facebook4j.FriendList>
getFriendRequests	friendRequests	[reading],[userId]	facebook4j.ResponseList <facebook4j.FriendRequest>
getFriends	friends	[reading],[userId]	facebook4j.ResponseList

Endpoint	Short Name	Options	Body Type
			<facebook4j.Friend>
getMutualFriends	mutualFriends	[friendUserId], [reading], [userId1,userId2]	facebook4j.ResponseList <facebook4j.Friend>
removeFriendlistMember	removeFriendlistMember	friendlistId, userId	boolean
Games			
deleteAchievement	deleteAchievement	achievementURL, [userId]	boolean
deleteScore	deleteScore	[userId]	boolean
getAchievements	achievements	[reading],[userId]	facebook4j.ResponseList <facebook4j.Achievement>
getScores	scores	[reading],[userId]	facebook4j.ResponseList <facebook4j.Score>
postAchievement	postAchievement	achievementURL, [userId]	String
postScore	postScore	scoreValue,[userId]	String
Groups			
getGroup	group	groupId,[reading]	facebook4j.Group
getGroupDocs	groupDocs	groupId,[reading]	facebook4j.ResponseList <facebook4j.GroupDoc>
getGroupFeed	groupFeed	groupId,[reading]	facebook4j.ResponseList <facebook4j.Post>
getGroupMembers	groupMembers	groupId,[reading]	facebook4j.ResponseList <facebook4j.GroupMember>
getGroupPictureURL	groupPictureURL	groupId	java.net.URL
getGroups	groups	[reading],[userId]	facebook4j.ResponseList <facebook4j.Group>
postGroupFeed	postGroupFeed	groupId, postUpdate	String
postGroupLink	postGroupLink	groupId,link, [message]	String
postGroupStatusMessage	postGroupStatusMessage	groupId, message	String
Insights			
getInsights	insights	objectId,metric, [reading]	facebook4j.ResponseList <facebook4j.Insight>
Likes			
getUserLikes	userLikes	[reading],[userId]	facebook4j.ResponseList <facebook4j.Like>
Links			
commentLink	commentLink	linkId,message	String
getLink	link	linkId,[reading]	facebook4j.Link
getLinkComments	linkComments	linkId,[reading]	facebook4j.ResponseList <facebook4j.Comment>
getLinkLikes	linkLikes	linkId,[reading]	facebook4j.ResponseList <facebook4j.Like>
likeLink	likeLink	linkId	boolean

Endpoint	Short Name	Options	Body Type
unlikeLink	unlikeLink	linkId	boolean
Locations			
getLocations	locations	[reading],[userId]	facebook4j.ResponseList <facebook4j.Location>
Messages			
getInbox	inbox	[reading],[userId]	facebook4j.InboxResponseList <facebook4j.Inbox>
getMessage	message	messageId,[reading]	facebook4j.Message
getOutbox	outbox	[reading],[userId]	facebook4j.ResponseList <facebook4j.Message>
getUpdates	updates	[reading],[userId]	facebook4j.ResponseList <facebook4j.Message>
Notes			
commentNote	commentNote	noteId,message	String
createNote	createNote	subject,message, [userId]	String
getNote	note	noteId,[reading]	facebook4j.Note
getNoteComments	noteComments	noteId,[reading]	facebook4j.ResponseList <facebook4j.Comment>
getNoteLikes	noteLikes	noteId,[reading]	facebook4j.ResponseList <facebook4j.Like>
getNotes	notes	[reading],[userId]	facebook4j.ResponseList <facebook4j.Note>
likeNote	likeNote	noteId	boolean
unlikeNote	unlikeNote	noteId	boolean
Notifications			
getNotifications	notifications	[includeRead], [reading],[userId]	facebook4j.ResponseList <facebook4j.Notification>
markNotificationAsRead	markNotificationAsRead	notificationId	boolean
Permissions			
getPermissions	permissions	[userId]	java.util.List <facebook4j.Permission>
revokePermission	revokePermission	permissionName, [userId]	boolean
Photos			
addTagToPhoto	addTagToPhoto	photoId,[toUserId], [toUserIds], [tagUpdate]	boolean
commentPhoto	commentPhoto	photoId,message	String
deletePhoto	deletePhoto	photoId	boolean
getPhoto	photo	photoId,[reading]	facebook4j.Photo
getPhotoComments	photoComments	photoId,[reading]	facebook4j.ResponseList <facebook4j.Comment>
getPhotoLikes	photoLikes	photoId,[reading]	facebook4j.ResponseList

Endpoint	Short Name	Options	Body Type
			<facebook4j.Like>
getPhotos	photos	[reading],[userId]	facebook4j.ResponseList <facebook4j.Photo>
getPhotoURL	photoURL	photoId	java.net.URL
getTagsOnPhoto	tagsOnPhoto	photoId,[reading]	facebook4j.ResponseList <facebook4j.Tag>
likePhoto	likePhoto	photoId	boolean
postPhoto	postPhoto	source,[message], [place],[noStory], [userId]	String
unlikePhoto	unlikePhoto	photoId	boolean
updateTagOnPhoto	updateTagOnPhoto	photoId,[toUserId], [tagUpdate]	boolean
Pokes			
getPokes	pokes	[reading],[userId]	facebook4j.ResponseList <facebook4j.Poke>
Posts			
commentPost	commentPost	postId,message	String
deletePost	deletePost	postId	boolean
getFeed	feed	[reading],[userId]	facebook4j.ResponseList <facebook4j.Post>
getHome	home	[reading]	facebook4j.ResponseList <facebook4j.Post>
getLinks	links	[reading],[userId]	facebook4j.ResponseList <facebook4j.Link>
getPost	post	postId,[reading]	facebook4j.Post
getPostComments	postComments	postId,[reading]	facebook4j.ResponseList <facebook4j.Comment>
getPostLikes	postLikes	postId,[reading]	facebook4j.ResponseList <facebook4j.Like>
getPosts	posts	[reading],[userId]	facebook4j.ResponseList <facebook4j.Post>
getStatuses	statuses	[reading],[userId]	facebook4j.ResponseList <facebook4j.Post>
getTagged	tagged	[reading],[userId]	facebook4j.ResponseList <facebook4j.Post>
likePost	likePost	postId	boolean
postFeed	postFeed	postUpdate,[userId]	String
postLink	postLink	link,[message], [userId]	String
postStatusMessage	postStatusMessage	message,[userId]	String
unlikePost	unlikePost	postId	boolean
Questions			
addQuestionOption	addQuestionOption	questionId, optionDescription	String

Endpoint	Short Name	Options	Body Type
createQuestion	createQuestion	question,[options], [allowNewOptions], [userId]	String
deleteQuestion	deleteQuestion	questionId	boolean
getQuestion	question	questionId,[reading]	facebook4j.Question
getQuestionOptions	questionOptions	questionId,[reading]	facebook4j.ResponseList <facebook4j.Question.Option>
getQuestionOptionVotes	questionOptionVotes	questionId	facebook4j.ResponseList <facebook4j.Question.Votes>
getQuestions	questions	[reading],[userId]	facebook4j.ResponseList <facebook4j.Question>
getSubscribedto	subscribedto	[reading],[userId]	facebook4j.ResponseList <facebook4j.Subscribedto>
getSubscribers	subscribers	[reading],[userId]	facebook4j.ResponseList <facebook4j.Subscriber>
Test Users			
createTestUser	createTestUser	appId,[name], [userLocale], [permissions]	facebook4j.TestUser
deleteTestUser	deleteTestUser	testUserId	boolean
getTestUsers	testUsers	appId	java.util.List <facebook4j.TestUser>
makeFriendTestUser	makeFriendTestUser	testUser1, testUser2	boolean
Users			
getMe	me	[reading]	facebook4j.User
getPictureURL	pictureURL	[size],[userId]	java.net.URL
getUser	user	userId,[reading]	facebook4j.User
getUsers	users	ids	java.util.List <facebook4j.User>
Videos			
commentVideo	commentVideo	videoId,message	String
getVideo	video	videoId,[reading]	facebook4j.Video
getVideoComments	videoComments	videoId,[reading]	facebook4j.ResponseList <facebook4j.Comment>
getVideoCover	videoCover	videoId	java.net.URL
getVideoLikes	videoLikes	videoId,[reading]	facebook4j.ResponseList <facebook4j.Like>
getVideos	videos	[reading],[userId]	facebook4j.ResponseList <facebook4j.Video>
likeVideo	likeVideo	videoId	boolean
postVideo	postVideo	source, [title,description], [userId]	String
unlikeVideo	unlikeVideo	videoId	boolean
Search			

Endpoint	Short Name	Options	Body Type
search	search	query,[reading]	facebook4j.ResponseList<facebook4j.internal.org.json.JSONObject>
searchCheckins	checkins	[reading]	facebook4j.ResponseList<facebook4j.Checkin>
searchEvents	events	query,[reading]	facebook4j.ResponseList<facebook4j.Event>
searchGroups	groups	query,[reading]	facebook4j.ResponseList<facebook4j.Group>
searchLocations	locations	[center,distance],[reading],[placeId]	facebook4j.ResponseList<facebook4j.Location>
searchPlaces	places	query,[reading],[center,distance]	facebook4j.ResponseList<facebook4j.Place>
searchPosts	posts	query,[reading]	facebook4j.ResponseList<facebook4j.Post>
searchUsers	users	query,[reading]	facebook4j.ResponseList<facebook4j.User>

3.20.4. Consumer Endpoints:

Any of the producer endpoints that take a [reading#reading](#) parameter can be used as a consumer endpoint. The polling consumer uses the **since** and **until** fields to get responses within the polling interval. In addition to other reading fields, an initial **since** value can be provided in the endpoint for the first poll.

Rather than the endpoints returning a List (or **facebook4j.ResponseList**) through a single route exchange, camel-facebook creates one route exchange per returned object. As an example, if "**facebook://home**" results in five posts, the route will be executed five times (once for each Post).

1. URI Options

Name	Type	Description
achievementURL	java.net.URL	The unique URL of the achievement
albumCreate	facebook4j.AlbumCreate	The facebook Album to be created
albumId	String	The album ID
allowNewOptions	boolean	True if allows other users to add new options
appId	String	The ID of the Facebook Application
center	facebook4j.GeoLocation	Location latitude and longitude
checkinCreate	facebook4j.CheckinCreate	The checkin to be created. Deprecated , instead create a Post with an attached location
checkinId	String	The checkin ID
commentId	String	The comment ID
description	String	The description text
distance	int	Distance in meters
domainId	String	The domain ID
domainName	String	The domain name
domainNames	String[]	The domain names
eventId	String	The event ID

Name	Type	Description
eventUpdate	facebook4j.EventUpdate	The event to be created or updated
friendId	String	The friend ID
friendUserId	String	The friend user ID
friendlistId	String	The friend list ID
friendlistName	String	The friend list Name
groupId	String	The group ID
ids	String[]	The ids of users
includeRead	boolean	Enables notifications that the user has already read in addition to unread ones
link	java.net.URL	Link URL
linkId	String	The link ID
locale	java.util.Locale	Desired FQL locale
message	String	The message text
messageId	String	The message ID
metric	String	The metric name
name	String	Test user name, must be of the form 'first last'
noStory	boolean	If set to true, optionally suppresses the feed story that is automatically generated on a user's profile when they upload a photo using your application.
noteId	String	The note ID
notificationId	String	The notification ID
objectId	String	The insight object ID
optionDescription	String	The question's answer option description
options	java.util.List<String>	The question's answer options
permissionName	String	The permission name
permissions	String	Test user permissions in the format perm1,perm2,...
photoId	String	The photo ID
place	String	The Facebook ID of the place associated with the Photo
placeId	String	The place ID
postId	String	The post ID
postUpdate	facebook4j.PostUpdate	The post to create or update
queries	java.util.Map<String>	FQL queries
query	String	FQL query or search terms for search* endpoints
question	String	The question text
questionId	String	The question id
reading	facebook4j.Reading	Optional reading parameters. See Reading Options(#reading)
scoreValue	int	The numeric score with value
size	facebook4j.PictureSize	The picture size, one of large, normal, small or square
source	facebook4j.Media	The media content from either a java.io.File or java.io.InputStream
subject	String	The note of the subject
tagUpdate	facebook4j.TagUpdate	Photo tag information
testUser1	facebook4j.TestUser	Test user
testUser2	facebook4j.TestUser	Test user
testUserId	String	The ID of the test user
title	String	The title text
toUserId	String	The ID of the user to tag
toUserIds	java.util.List<String>	The IDs of the users to tag

Name	Type	Description
userId	String	The Facebook user ID
userId1	String	The ID of a user
userId2	String	The ID of a user
userIds	String[]	The IDs of users to invite to event
userLocale	String	The test user locale
videoId	String	The video ID

3.20.5. Reading Options

The **reading** option of type **facebook4j.Reading** adds support for reading parameters, which allow selecting specific fields, limits the number of results, etc. For more information see [Graph API#reading - Facebook Developers](#).

It is also used by consumer endpoints to poll Facebook data to avoid sending duplicate messages across polls.

The reading option can be a reference or value of type **facebook4j.Reading**, or can be specified using the following reading options in either the endpoint URI or exchange header with **CamelFacebook.** prefix.

Option	Description
reading.fields	Field names to retrieve, in the format field1,field2,...
reading.limit	Limit for number of items to return for list results, e.g. a limit of 10 returns items 1 through 10
reading.offset	Starting offset for list results, e.g. a limit of 10, and offset of 10 returns items 11 through 20
reading.until	A Unix timestamp or strptime data value that points to the end of the range of time-based data
reading.since	A Unix timestamp or strptime data value that points to the start of the range of time-based data
reading.locale	Retrieve localized content in a particular locale, specified as a String with the format language[,country][,variant]
reading.with	Retrieve information about objects that have location information attached, set it to true
reading.metadata	Use Facebook Graph API Introspection to retrieve object metadata, set it to true
reading.filter	User's stream filter key. See https://developers.facebook.com/docs/technical-guides/fql

3.20.6. Message header

Any of the [URI options#urioptions](#) can be provided in a message header for producer endpoints with **CamelFacebook.** prefix.

3.20.7. Message body

All result message bodies utilize objects provided by the Facebook4J API. Producer endpoints can specify the option name for incoming message body in the **inBody** endpoint parameter.

For endpoints that return an array, or **facebook4j.ResponseList**, or **java.util.List**, a consumer endpoint will map every elements in the list to distinct messages.

3.20.8. Use cases

To create a post within your Facebook profile, send this producer a **facebook4j.PostUpdate** body.

```
from("direct:foo")
  .to("facebook://postFeed/inBody=postUpdate");
```

To poll, every 5 sec (You can set the [polling consumer](#) options by adding a prefix of "consumer"), all statuses on your home feed:

```
from("facebook://home?consumer.delay=5000")
  .to("bean:blah");
```

Searching using a producer with dynamic options from header.

In the bar header we have the Facebook search string we want to execute in public posts, so we need to assign this value to the CamelFacebook.query header.

```
from("direct:foo")
  .setHeader("CamelFacebook.query", header("bar"))
  .to("facebook://posts");
```

3.21. File

The File component provides access to file systems, allowing files to be processed by any other Camel [Components](#) or messages from other components to be saved to disk.

3.21.1. 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&...



Camel supports only endpoints configured with a starting directory. So the **directoryName** must be a directory. If you want to consume a single file only, you can use the *fileName* option, e.g. by setting *fileName=thefilename*. Also, the starting directory must not contain dynamic expressions with `${ }` placeholders. Again use the *fileName* option to specify the dynamic part of the filename.



You need to 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 *doneFileName* option that you can use. See also the section *Consuming files from folders where others drop files directly*.*

3.21.2. URI Options

3.21.2.1. 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	<p>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 takes precedence over the CamelFileName header. (Note: The header itself can also be an Expression).</p> <p>The expression options support both String and Expression types. If the expression is a String type, it is always evaluated using the File Language.</p> <p>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. Starting with Camel 2.11, the producers support the CamelOverrideFileName header which will take precedence over any existing CamelFileName header. CamelOverrideFileName is used only once, and helps avoid temporarily storing a CamelFileName and needing to restore it afterwards.</p>
flatten	false	<p>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.</p> <p>Setting this to true on the producer ensures that any file name received in CamelFileName header will be stripped of any leading paths.</p>
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. You can use this on the consumer, to specify the encodings of the files, which allow Camel to know the charset it should load the file content in case the file content is being accessed. Likewise when writing a file, you can use this option to specify which charset to write the file as well.
copyAndDeleteOnRenameFail	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.
renameUsingCopy	false	Perform rename operations using a copy and delete strategy. This is primarily used in environments where the regular rename operation is unreliable (e.g. across different file systems or networks). This option takes precedence over the copyAndDeleteOnRenameFail parameter that will automatically fall back to the copy and delete strategy, but only after additional delays.

3.21.2.2. 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.
runLoggingLevel	TRACE	The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.

Name	Default Value	Description
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 <code>noop=true</code> , Camel will set <code>idempotent=true</code> 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 <code>order</code> .
move	<code>.camel</code>	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 <code>.done</code> subdirectory just enter <code>.done</code> .
moveFailed	null	Expression (such as File Language) used to dynamically set a different target directory when moving files in case of processing (configured via <code>move</code> setting defined above) failed. For example, to move files into a <code>.error</code> subdirectory use: <code>.error</code> . 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 <code>{{antInclude=**/*.txt}}</code> . Multiple inclusions may be specified in comma-delimited format.
antExclude	null	Ant style filter exclusion. If both <code>antInclude</code> and <code>antExclude</code> the latter takes precedence. Multiple exclusions may be specified in comma-delimited format.
antFilter-CaseSensitive	true	Starting with Camel 2.11, whether Ant filters are case sensitive or not.
idempotent	false	Option to use the Idempotent Consumer EIP pattern to let Camel skip already processed files. This will by default use a memory based LRU Cache that holds 1000 entries. If <code>noop=true</code> then <code>idempotent</code> will be enabled as well to avoid consuming the same files over and over again.
idempotentKey	Expression	Starting with Camel 2.11, use of a custom idempotent key. By default the absolute path of the file will be used. Camel's File Language can be used to specify the file name and size: <code>idempotentKey=\${file:name}-\${file:size}</code> .
idempotent-Repository	null	Pluggable repository as a <code>org.apache.camel.processor.idempotent.MessageIdRepository</code> class. This will by default use <code>MemoryMessageIdRepository</code> if none is specified and <code>idempotent</code> 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 <code>org.apache.camel.component.file.GenericFileFilter</code> class. This will skip files if filter returns <code>false</code> in its <code>accept()</code> method.
sorter	null	Pluggable sorter as a java.util.Comparator <code><org.apache.camel.component.file.GenericFile></code> 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.
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 build in strategies: markerFile Camel creates a marker file (<code>fileName.camelLock</code>) and then holds a lock on it. This option is <code>*not*</code> available 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. The option <code>readLockCheckInterval</code> can be used to set the

Name	Default Value	Description
		<p>check frequency. Note the FTP option <code>fastExistsCheck</code> can be enabled to speed up this <code>readLock</code> strategy, if the FTP server supports the LIST operation with a full file name (some servers may not). not avail for the FTP component.</p> <p><code>fileLock</code> is for using <code>java.nio.channels.FileLock</code>. This option is not available for the FTP component. This approach should be avoided when accessing a remote file system via a mount/share unless that file system supports distributed file locks.</p> <p><code>rename</code> is for using a try to rename the file as a test if we can get exclusive read-lock.</p> <p><code>none</code> is for no read locks at all. Note the read locks changed, <code>fileLock</code> and <code>rename</code> will also use a <code>markerFile</code> as well, to ensure not picking up files that may be in process by another Camel consumer running on another node (eg cluster). This is supported only by the file component (not the ftp component).</p>
<code>readLockTimeout</code>	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 <code>fileLock</code> , <code>changed</code> and <code>rename</code> support the timeout. Note: for the FTP component the default value is 20000.
<code>readLockCheck-Interval</code>	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 <code>changed</code> read lock, you can set a higher interval period to cater for <i>slow writes</i> . The default of 1 sec. may be <i>too fast</i> if the producer is very slow writing the file.
<code>readLock-MinLength</code>	1	This option applied only for <code>readLock=changed</code> . This option allows you to configure a minimum file length. By default Camel expects the file to contain data, and thus the default value is 1. You can set this option to zero to allow consuming zero-length files.
<code>readLockLoggingLevel</code>	WARN	Starting with Camel 2.12: Logging level used when a read lock could not be acquired. By default a WARN is logged. You can change this level, for example to OFF to not have any logging. This option is only applicable for <code>readLock</code> of types: <code>changed</code> , <code>fileLock</code> , <code>rename</code> .
<code>directoryMust-Exist</code>	false	Similar to <code>startingDirectoryMustExist</code> but this applies during polling recursive sub-directories.
<code>doneFileName</code>	null	If provided, Camel will only consume files if a <i>done</i> file exists. This option configures what file name to use. Either you can specify a fixed name, or you can use dynamic placeholders. The <i>done</i> file is always expected in the same folder as the original file. See <i>using done file</i> and <i>writing done file</i> sections for examples.
<code>exclusiveRead-LockStrategy</code>	null	Pluggable read-lock as a <code>org.apache.camel.component.file.GenericFileExclusiveReadLockStrategy</code> implementation.
<code>maxMessages-PerPoll</code>	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 <code>eagerMaxMessagesPerPoll</code> option and set this to <code>false</code> to allow to scan all files first and then sort afterwards.
<code>eagerMax-MessagesPerPoll</code>	true	Allows for controlling whether the limit from <code>maxMessagesPerPoll</code> is eager or not. If eager then the limit is during the scanning of files. Whereas <code>false</code> would scan all files, and then perform sorting. Setting this option to <code>false</code> 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.
<code>minDepth</code>	0	The minimum depth to start processing when recursively processing a directory. Using <code>minDepth=1</code> means the base directory. Using <code>minDepth=2</code> means the first sub directory.
<code>maxDepth</code>	Integer. MAX_VALUE	The maximum depth to traverse when recursively processing a directory.
<code>processStrategy</code>	null	A pluggable <code>org.apache.camel.component.file.GenericFileProcessStrategy</code> allowing you to implement your own <code>readLock</code> option or similar. Can also be used when special conditions must be

Name	Default Value	Description
		met before a file can be consumed, such as a special <i>ready</i> file exists. If this option is set then the <code>readLock</code> option does not apply.
<code>startingDirectoryMustExist</code>	false	whether the starting directory must exist. Keep in mind that the <code>autoCreate</code> option is default enabled, which means the starting directory is normally auto created if it doesn't exist. You can disable <code>autoCreate</code> and enable this to ensure the starting directory must exist. It will then throw an exception if the directory doesn't exist.
<code>pollStrategy</code>	null	A pluggable <code>org.apache.camel.spi.PollingConsumerPollingStrategy</code> 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.
<code>sendEmptyMessageWhenIdle</code>	false	If the polling consumer did not poll any files, you can enable this option to send an empty message (no body) instead.
<code>consumer.bridgeErrorHandler</code>	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 <code>org.apache.camel.spi.ExceptionHandler</code> to deal with exceptions, that by default will be logged at WARN/ERROR level and ignored.
<code>scheduled-ExecutorService</code>	null	Allows for configuring a custom/shared thread pool to use for the consumer. By default each consumer has its own single threaded thread pool. This option allows you to share a thread pool among multiple file consumers.
<code>scheduler</code>	null	Camel 2.12: To use a custom scheduler to trigger the consumer to run. See more details at Polling Consumer , for example there is a Quartz2 , and Spring based scheduler that supports CRON expressions.
<code>backoffMultiplier</code>	0	Camel 2.12: To let the scheduled polling consumer backoff if there has been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt is happening again. When this option is in use then <code>backoffIdleThreshold</code> and/or <code>backoffErrorThreshold</code> must also be configured. See more details at Polling Consumer .
<code>backoffIdleThreshold</code>	0	Camel 2.12: The number of subsequent idle polls that should happen before the <code>backoffMultiplier</code> should kick-in.
<code>backoffErrorThreshold</code>	0	Camel 2.12: The number of subsequent error polls (failed due some error) that should happen before the <code>backoffMultiplier</code> should kick-in.

3.21.2.3. 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.

3.21.2.4. Producer

Name	Default Value	Description
<code>fileExist</code>	Override	<p>What to do if a file already exists with the same name. The following values can be specified: Override, Append, Fail, Ignore, Move, and TryRename (Camel 2.11.1).</p> <ul style="list-style-type: none"> • Override, which is the default, replaces the existing file.

Name	Default Value	Description
		<ul style="list-style-type: none"> Append adds content to the existing file. Fail throws a <code>GenericFileOperation-Exception</code>, 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. The Move option will move any existing files, before writing the target file. The corresponding <code>moveExisting</code> option must be configured. The option <code>eagerDeleteTargetFile</code> can be used to control what to do if an moving the file, and there exists already an existing file, otherwise causing the move operation to fail. TryRename (<i>Camel 2.11.1</i>) is only applicable if <code>tempFileName</code> option is in use. This allows to try renaming the file from the temporary name to the actual name, without doing any exists check. This check may be faster on some file systems and especially FTP servers.
<code>tempPrefix</code>	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.
<code>tempFileName</code>	null	The same as <code>tempPrefix</code> option but offering a more fine grained control on the naming of the temporary filename as it uses the File Language .
<code>moveExisting</code>	null	Expression used to compute file name to use when <code>fileExist=Move</code> is configured. To move files into a backup subdirectory just enter backup. This option supports only the following File Language tokens: "file:name", "file:name.ext", "file:name.noext", "file:onlyname", "file:onlyname.noext", "file:ext", and "file:parent". Notice the "file:parent" is not supported by the FTP component, as the FTP component can move existing files only to a relative directory based on the current directory.
<code>keepLastModified</code>	false	If enabled, will keep the last modified timestamp from the source file (if any). This will use the Exchange. <code>FILE_LAST_MODIFIED</code> header to located the timestamp. This header can contain either a <code>java.util.Date</code> 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 <i>cannot</i> use this option with any of the ftp producers.
<code>eagerDeleteTarget-File</code>	true	Whether or not to <i>eagerly delete</i> any existing target file. (This option only applies when you use <code>fileExists=Override</code> and the <code>tempFileName</code> 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 <code>eagerDeleteTargetFile</code> 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. This option is also used to control whether to delete any existing files when <code>fileExist=Move</code> is enabled and an existing file is present. If this option <code>copyAndDeleteOnRenameFail</code> is false, then an exception will be thrown if an existing file existed, if it's true, then the existing file is deleted before the move operation.
<code>doneFileName</code>	null	If provided, then Camel will write a second <i>done</i> file when the original file has been written. The <i>done</i> 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 <i>done</i> file will always be written in the same folder as the original file. See <i>writing done file</i> section for examples.
<code>allowNullBody</code>	false	Used to specify if a null body is allowed during file writing. If set to true then an empty file will be created, when set to false, and attempting to send a null body to the file component, a <code>GenericFileWriteException</code> of "Cannot write null body to file" will be thrown. If the "fileExist" option is set to "Override", then the file will be truncated, and if set to "append" the file will remain unchanged.
<code>forceWrites</code>	true	Starting with Camel 2.10.5/2.11, whether to force syncing writes to the file system. You can turn this off if you do not want this level of guarantee, for example if writing to logs / audit logs etc; this would yield better performance.

3.21.2.5. Default behavior for file producer

By default it will override any existing file, if one exist with the same name.



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.

3.21.3. 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 is considered as a directory name (though if you use an expression such as [File Language](#), or [Simple](#) then the result of the expression evaluation is the file name to be used - eg if you set

```
move=../backup/copy-of-${file:name}
```

then that's using the [File Language](#) which we use return the file name to be used), 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://inbox?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.

3.21.3.1. 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}
```

3.21.3.2. About moveFailed

The `moveFailed` option allows you to move files that **could not** be processed successfully 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/${file:name.next}-${date:now:yyyyMMddHHmmssSSS}.${file:ext}`.

See more examples in [File Language](#)

3.21.4. Message Headers

The following headers are supported by this component:

3.21.4.1. File producer only

Header	Description
<code>CamelFileName</code>	Specifies the name of the file to write (relative to the endpoint directory). The name can be a <code>String</code> ; a <code>String</code> with a File Language or Simple expression; or an Expression object. If it is <code>null</code> then Camel will auto-generate a filename based on the message unique ID.
<code>CamelFileNameProduced</code>	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.
<code>CamelOverruleFileName</code>	Starting with Camel 2.11, this field is used for overruling <code>CamelFileName</code> header and use the value instead (but only once, as the producer will remove this header after writing the file). The value can be only be a <code>String</code> . Notice that if the option <code>fileName</code> has been configured, then this is still being evaluated.

3.21.4.2. File consumer only

Header	Description
<code>CamelFileName</code>	Name of the consumed file as a relative file path with offset from the starting directory configured on the endpoint.
<code>CamelFileNameOnly</code>	Just the file name (the name with no leading paths).
<code>CamelFileAbsolute</code>	A <code>boolean</code> option specifying whether the consumed file denotes an absolute path or not. It should normally be <code>false</code> 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.
<code>CamelFileAbsolutePath</code>	The absolute path to the file. For relative files this path holds the relative path instead.

Header	Description
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 long value containing the last modified timestamp of the file. In Camel 2.10.3 and older the type is Date.

3.21.5. Batch Consumer

This component implements the [Batch Consumer](#).

3.21.5.1. 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 [Aggregator](#) aggregate that precise number of files.

3.21.6. 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");
```

3.21.7. 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.

3.21.8. 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.

3.21.9. Using done files

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. For versions prior to 2.9.3, Camel will automatically 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

3.21.10. Writing done files

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.

3.21.11. Samples

3.21.11.1. Read from a directory and write to another directory

```
from("file://inputdir/?delete=true").to("file://outputdir")
```

The above will listen on a directory and create a message for each file dropped there. It will copy the contents to the `outputdir` and delete the file in the `inputdir`. Using `.to("file://outputdir?overrideFile=copy-of-${file:name}")` instead will allow you to write to another directory using a dynamic override name.

Read from a directory and write to another directory using a override dynamic name

```
from("file://inputdir/?delete=true").to("file://outputdir?overrideFile=copy-of-${file:name}")
```

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`.

3.21.11.2. 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
```

3.21.11.3. 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
```

3.21.11.4. 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.

3.21.11.5. 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");
    }
}

```

3.21.11.6. 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.

3.21.11.7. Writing file through the temporary directory relative to the final destination

Sometime you need to temporarily write the files to some directory relative to the destination directory. Such situation usually happens when some external process with limited filtering capabilities is reading from the directory you are writing to. In the example below files will be written to the `/var/myapp/filesInProgress` directory and after data transfer is done, they will be atomically moved to the `/var/myapp/finalDirectory` directory.

```
from("direct:start").
  to("file:///var/myapp/finalDirectory?tempPrefix=../filesInProgress/");
```

3.21.11.8. 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.

3.21.12. Avoiding reading the same file more than once (idempotent consumer)

Camel supports *[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("../");
```

Camel uses the absolute file name as the idempotent key, to detect duplicate files. From Camel 2.11 onwards you can customize this key by using an expression in the `idempotentKey` option. For example to use both the name and the file size as the key:

```
<route>
  <from
    uri="file://inbox?idempotent=true&idempotentKey=${file:name}-${file-size}" />
  <to uri="bean:processInbox" />
</route>
```

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 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&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
```

3.21.13. 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>
```

3.21.13.1. Filtering using ANT path matcher



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>
```

3.21.14. 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`.

3.21.15. 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:modified
```

Then we want to group by name as a second option so files with same modification is sorted by name:

```
sortBy=file:modified;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 `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
```

3.21.16. 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 `abort()` special logic can be executed in case the `begin` operation returned false, for example to cleanup resources, etc.
- in the `commit()` method we can move the file and also delete the *ready* file.

3.22. Flatpack

3.22.1. Flatpack Component

The Flatpack component supports fixed width and delimited file parsing via the [FlatPack library](#).



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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.22.1.1. 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&...`

3.22.1.2. 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.
allowShortLines	false	Camel 2.9.7 and 2.10.5 onwards: Allows for lines to be shorter than expected and ignores the extra characters.
ignoreExtraColumns	false	Camel 2.9.7 and 2.10.5 onwards: Allows for lines to be longer than expected and ignores the extra characters.

3.22.1.3. 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.

3.22.1.4. Message Headers

Camel will store the following headers on the IN message:

Header	Description
<code>camelFlatpackCounter</code>	The current row index. For <code>splitRows=false</code> the counter is the total number of rows.

3.22.1.5. 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");
```

3.22.1.6. 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>
```

3.22.1.7. 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](#)

3.22.2. Flatpack DataFormat

The *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 [Splitter](#).

Notice: The Flatpack library does currently not support header and trailers for the marshal operation.

3.22.2.1. 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)

Option	Default	Description
parserFactory	null	Uses the default Flatpack parser factory.

3.22.2.2. Usage

To use the data format, simply instantiate an instance and invoke the marshal or unmarshal operation in the Route designer:

```
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.

3.22.2.3. 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>
```

3.23. FOP

The FOP component allows you to render a message into different output formats using [Apache FOP](#).

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-fop</artifactId>
  <version>x.x.x</version>
```

```
<!-- use the same version as your Camel core version -->
</dependency>
```

3.23.1. URI format

```
fop://outputFormat?[options]
```

3.23.2. Output Formats

The primary output format is PDF but other output [formats](#) are also supported:

name	outputFormat	description
PDF	application/pdf	Portable Document Format
PS	application/postscript	Adobe Postscript
PCL	application/x-pcl	Printer Control Language
PNG	image/png	PNG images
JPEG	image/jpeg	JPEG images
SVG	image/svg+xml	Scalable Vector Graphics
XML	application/X-fop-areatree	Area tree representation
MIF	application/mif	FrameMaker's MIF
RTF	application/rtf	Rich Text Format
TXT	text/plain	Text

3.23.3. Endpoint Options

Name	Default Value	Description
userConfigURL	none	The location of a configuration file with the following structure . From Camel 2.12 onwards the file is loaded from the classpath by default. You can use <code>file:</code> , or <code>classpath:</code> as prefix to load the resource from file or classpath. In previous releases the file is always loaded from file system.

3.23.4. Message Operations

name	default value	description
CamelFop.Output.Format		Overrides the output format for that message
CamelFop.Encrypt.userPassword		PDF user password
CamelFop.Encrypt.ownerPassword		PDF owner password
CamelFop.Encrypt.allowPrint	true	Allows printing the PDF
CamelFop.Encrypt.allowCopyContent	true	Allows copying content of the PDF
CamelFop.Encrypt.allowEditContent	true	Allows editing content of the PDF
CamelFop.Encrypt.allowEditAnnotations	true	Allows editing annotation of the PDF
CamelFop.Render.producer	Apache FOP	Metadata element for the system/software that produces the document

name	default value	description
CamelFop.Render.creator		Metadata element for the user that created the document
CamelFop.Render.creationDate		Creation Date
CamelFop.Render.author		Author of the content of the document
CamelFop.Render.title		Title of the document
CamelFop.Render.subject		Subject of the document
CamelFop.Render.keywords		Set of keywords applicable to this document

3.23.5. Example

Below is an example route that renders PDFs from xml data and xslt template and saves the PDF files in target folder:

```
from("file:source/data/xml")
  .to("xslt:xslt/template.xsl")
  .to("fop:application/pdf")
  .to("file:target/data");
```

3.24. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.24.1. 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&...`

3.24.2. 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 <code>clearContentCache</code> operation.

Option	Default	Description
encoding	null	Character encoding of the resource content.
templateUpdateDelay	5	Character encoding of the resource content.

3.24.3. 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`.

3.24.4. Freemarker Context

Camel will provide exchange information in the Freemarker context (just a `Map`). The `Exchange` is transferred as:

key	value
<code>exchange</code>	The Exchange itself.
<code>exchange.properties</code>	The Exchange properties.
<code>headers</code>	The headers of the In message.
<code>camelContext</code>	The Camel Context.
<code>request</code>	The In message.
<code>body</code>	The In message body.
<code>response</code>	The Out message (only for InOut message exchange pattern).

3.24.5. 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.

3.24.6. 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.

3.24.7. 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");
```

3.25. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```



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.



This component uses two different libraries for the FTP work. FTP and FTPS uses [Apache Commons Net](#) while SFTP uses [JCraft JSCH](#).

3.25.1. URI format and Options

```
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&...`, where *option* can be:

Table 3.9.

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 (<code>false</code>).
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 (<code>false</code>).
securityProtocol	TLS	FTPS only : Sets the underlying security protocol. The following values are defined: <code>TLS</code> : Transport Layer Security <code>SSL</code> : Secure Sockets Layer
disableSecureData-ChannelDefaults	false	FTPS only : Whether or not to disable using default values for <code>execPbsz</code> and <code>execProt</code> when using secure data transfer. You can set this option to <code>true</code> if you want to be in absolute full control what the options <code>execPbsz</code> and <code>execProt</code> should be used.
download	true	Starting with Camel 2.11, whether the FTP consumer should download the file. If this option is set to <code>false</code> , then the message body will be null, but the consumer will still trigger a Camel Exchange that has details about the file such as file name, file size, etc. It's just that the file will not be downloaded.
streamDownload	false	Starting with Camel 2.11, whether the consumer should download the entire file up front, the default behavior, or if it should pass an <code>InputStream</code> read from the remote resource rather than an in-memory array as the in-body of the Camel Exchange. This option is ignored if <code>download</code> is <code>false</code> or if <code>localWorkDirectory</code> is provided. This option is useful for working with large remote files.
execProt	null	FTPS only : This will use option <code>P</code> by default, if secure data channel defaults hasn't been disabled. Possible values are: <code>C</code> : Clear <code>S</code> : Safe (SSL protocol only) <code>E</code> : Confidential (SSL protocol only) <code>P</code> : Private
execPbsz	null	FTPS only : This option specifies the buffer size of the secure data channel. If option <code>useSecureDataChannel</code> has been enabled and this option has not been explicit set, then value <code>0</code> is used.
isImplicit	false	FTPS only : Sets the security mode(implicit/explicit). Default is explicit (<code>false</code>).
knownHostsFile	null	SFTP only : Sets the <code>known_hosts</code> file, so that the SFTP endpoint can do host key verification.
knownHostsUri	null	SFTP only : Sets the <code>known_hosts</code> file (loaded from classpath by default), so that the SFTP endpoint can do host key verification.
keyPair	null	SFTP only : Sets the Java <code>KeyPair</code> for SSH public key authentication, it supports DSA or RSA keys.
privateKeyFile	null	SFTP only : Set the private key file to that the SFTP endpoint can do private key verification.

Name	Default Value	Description
privateKeyUri	null	SFTP only: Set the private key file (loaded from classpath by default) to that the SFTP endpoint can do private key verification.
privateKey	null	SFTP only: Set the private key as byte[] to that the SFTP endpoint can do private key verification.
privateKeyFilePassphrase	null	SFTP only: <i>Deprecated:</i> use privateKeyPassphrase instead. Set the private key file passphrase to that the SFTP endpoint can do private key verification.
privateKeyPassphrase	null	SFTP only: Set the private key file passphrase to that the SFTP endpoint can do private key verification.
preferredAuthentications	null	SFTP only: Set the preferred authentications which SFTP endpoint will used. Some example include:password,publickey. If not specified the default list from JSCH will be used.
ciphers	null	A comma separated list of ciphers that will be used in order of preference. Possible cipher names are defined by JCraft JSCH . Some examples include: aes128-ctr,aes128-cbc,3des-ctr,3des-cbc,blowfish-cbc,aes192-cbc,aes256-cbc. If not specified the default list from JSCH will be used.
fastExistsCheck	false	If true, camel-ftp will use the list file directly to check if the file exists. Since some FTP servers may not support listing the file directly, if the option is false, camel-ftp will use the old way to list the directory and check if the file exists. Note this option also influences readLock=changed to control whether it performs a fast check to update file information or not. This can be used to speed up the process if the FTP server has a lot of files.
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	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	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.

Name	Default Value	Description
<code>ftpClientConfig</code>	<code>null</code>	FTP and FTPS Only: Allows you to use a custom <code>org.apache.commons.net.ftp.FTPClientConfig</code> instance.
<code>ftpClient.trustStore.file</code>	<code>null</code>	FTPS Only: Sets the trust store file, so that the FTPS client can look up for trusted certificates.
<code>ftpClient.trustStore.type</code>	<code>JKS</code>	FTPS Only: Sets the trust store type.
<code>ftpClient.trustStore.algorithm</code>	<code>SunX509</code>	FTPS Only: Sets the trust store algorithm.
<code>ftpClient.trustStore.password</code>	<code>null</code>	FTPS Only: Sets the trust store password.
<code>ftpClient.keyStore.file</code>	<code>null</code>	FTPS Only: Sets the key store file, so that the FTPS client can look up for the private certificate.
<code>ftpClient.keyStore.type</code>	<code>JKS</code>	FTPS Only: Sets the key store type.
<code>ftpClient.keyStore.algorithm</code>	<code>SunX509</code>	FTPS Only: Sets the key store algorithm.
<code>ftpClient.keyStore.password</code>	<code>null</code>	FTPS Only: Sets the key store password.
<code>ftpClient.keyStore.keyPassword</code>	<code>null</code>	FTPS Only: Sets the private key password.
<code>sslContextParameters</code>	<code>null</code>	FTPS Only: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry . This reference overrides any configured SSL related options on <code>ftpClient</code> as well as the <code>securityProtocol</code> (SSL, TLS, etc.) set on <code>FtpsConfiguration</code> . See Using the JSSE Configuration Utility .
<code>proxy</code>	<code>null</code>	SFTP Only: Reference to a <code>com.jcraft.jsch.Proxy</code> in the Registry . This proxy is used to consume/send messages from the target SFTP host.
<code>useList</code>	<code>true</code>	FTP/FTPS Only: Whether the consumer should use FTP LIST command to retrieve directory listing to see which files exists. If this option is set to <code>false</code> , then <code>stepwise=false</code> must be configured, and also <code>fileName</code> must be configured to a fixed name, so the consumer knows the name of the file to retrieve. When doing this only that single file can be retrieved. See further below for more details.
<code>ignoreFileNotFoundOrPermissionError</code>	<code>false</code>	Whether the consumer should ignore when a file was attempted to be retrieved but did not exist (for some reason), or failure due insufficient file permission error.

Note besides those listed below, all options from the [File](#) are inherited and hence available to the FTP component.



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" );
```

3.25.2. More URI options



See [File](#) as all the options there also apply to this component.

3.25.3. 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 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. See the [online Camel documentation](#) for examples of both techniques.

3.25.4. 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
```



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.



This component is an extension of the [File](#) component, and there are more samples and details on the [File](#) component page.

3.25.5. 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.

3.25.5.1. 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.

3.25.6. Message Headers

The following message headers can be used to affect the behavior of the component

Header	Description
<code>CamelFileName</code>	Specifies the output file name (relative to the endpoint directory) to be used for the output message when sending to the endpoint. If neither <code>CamelFileName</code> or an expression are specified, then a generated message ID is used as the filename instead.
<code>CamelFileNameProduced</code>	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.
<code>CamelFileBatchIndex</code>	Current index out of total number of files being consumed in this batch.
<code>CamelFileBatchSize</code>	Total number of files being consumed in this batch.
<code>CamelFileHost</code>	The remote hostname.
<code>CamelFileLocalWorkPath</code>	Path to the local work file, if local work directory is used.

In addition the FTP/FTPS consumer and producer will enrich the Camel `Message` with the following headers.

Header	Description
<code>CamelFtpReplyCode</code>	Camel 2.11.1: The FTP client reply code (the type is a integer)
<code>CamelFtpReplyString</code>	Camel 2.11.1: The FTP client reply string

3.25.7. 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.

3.25.8. 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");
```



Renaming the work file facilitates optimization. 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.

3.25.9. 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&binary=true&delay=60000"/>

  <to uri="file://target/test-reports"/>
</route>
```

3.25.9.1. 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");
```

3.25.9.2. 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");
```

3.25.10. 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.

```
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:

```
<!-- define our filter as a plain Spring bean -->
<bean id="myFilter" class="com.mycompany.MyFileFilter"/>

<route>
    <from uri="ftp://someuser@someftpserver.com?password=secret    //
        &filter=#myFilter"/>
    <to uri="bean:processInbox"/>
</route>
```

3.25.11. 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:

```
<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>
```

3.25.12. Using a proxy with SFTP

To use an HTTP proxy to connect to your remote host, you can configure your route in the following way:

```
<!-- define our sorter as a plain spring bean -->
<bean id="proxy" class="com.jcraft.jsch.ProxyHTTP">
  <constructor-arg value="localhost"/>
  <constructor-arg value="7777"/>
</bean>

<route>
  <from uri="sftp://localhost:9999/root?username=admin&password=admin&proxy=#proxy"/>
  <to uri="bean:processFile"/>
</route>
```

You can also assign a user name and password to the proxy, if necessary. Please consult the documentation for `com.jcraft.jsch.Proxy` to discover all options.

3.25.13. Setting preferred SFTP authentication method

If you want to explicitly specify the list of authentication methods that should be used by `sftp` component, use `preferredAuthentications` option. If for example you would like Camel to attempt to authenticate with private/public SSH key and fallback to user/password authentication in the case when no public key is available, use the following route configuration:

```
from("sftp://localhost:9999/root?
username=admin&password=admin&preferredAuthentications=publickey,password").
```



```
to("bean:processFile");
```

3.25.14. Consuming a single file using a fixed name

When you want to download a single file and knows the file name, you can use `fileName=myFileName.txt` to tell Camel the name of the file to download. By default the consumer will still do a FTP LIST command to do a directory listing and then filter these files based on the `fileName` option. Though in this use-case it may be desirable to turn off the directory listing by setting `useList=false`. For example the user account used to login to the FTP server may not have permission to do a FTP LIST command. So you can turn off this with `useList=false`, and then provide the fixed name of the file to download with `fileName=myFileName.txt`, then the FTP consumer can still download the file. If the file for some reason does not exist, then Camel will by default throw an exception, you can turn this off and ignore this by setting `ignoreFileNotFoundOrPermissionError=true`.

For example to have a Camel route that pickup a single file, and delete it after use you can do

```
from("ftp://admin@localhost:21/nolist/?
password=admin&stepwise=false&useList=false&ignoreFileNotFoundOrPermissionError=
true&fileName=report.txt&delete=true")
.to("activemq:queue:report");
```

Notice that we have use all the options we talked above above.

You can also use this with `ConsumerTemplate`. For example to download a single file (if it exists) and grab the file content as a `String` type:

```
String data = template.retrieveBodyNoWait("ftp://admin@localhost:21/nolist/?
password=admin&stepwise=false&useList=false&ignoreFileNotFoundOrPermissionError=
true&fileName=report.txt&delete=true", String.class);
```

3.25.15. Debug logging

This component has log level **TRACE** that can be helpful if you have problems.

3.26. Geocoder

The **geocoder**: component is used for looking up geocodes (latitude and longitude) for a given address, or reverse lookup. The component uses the [Java API for Google Geocoder](#) library.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-geocoder</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.26.1. URI format

```
geocoder:address:name[?options]
```

```
geocoder:latlng:latitude,longitude[?options]
```

3.26.2. Options

Property	Default	Description
language	en	The language to use.
headersOnly	false	Whether to only enrich the Exchange with headers, and leave the body as-is.
clientId	en	To use google premium with this client id
clientKey	en	To use google premium with this client key

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.26.3. Exchange data format

Camel will deliver the body as a `com.google.code.geocoder.model.GeocodeResponse` type.

And if the address is "current" then the response is a String type with a JSON representation of the current location.

If the option `headersOnly` is set to `true` then the message body is left as-is, and only headers will be added to the [Exchange](#).

3.26.4. Message Headers

DataFormat	Description
CamelGeoCoderStatus	Mandatory. Status code from the geocoder library. If status is <code>GeocoderStatus.OK</code> then additional headers is enriched
CamelGeoCoderAddress	The formatted address
CamelGeoCoderLat	The latitude of the location.
CamelGeoCoderLng	The longitude of the location.
CamelGeoCoderLatLng	The latitude and longitude of the location. Separated by comma.
CamelGeoCoderCity	The city long name.
CamelGeoCoderRegionCode	The region code.
CamelGeoCoderRegionName	The region name.
CamelGeoCoderCountryLong	The country long name.
CamelGeoCoderCountryLong	The country long name.

Notice not all headers may be provided depending on available data and mode in use (address vs latlng).

3.26.5. Samples

In the example below we get the latitude and longitude for Paris, France

```
from("direct:start")
.to("geocoder:address:Paris, France")
```

If you provide a header with the `CamelGeoCoderAddress` then that overrides the endpoint configuration, so to get the location of Copenhagen, Denmark we can send a message with a headers as shown:

```
template.sendBodyAndHeader("direct:start", "Hello", GeoCoderConstants.ADDRESS,
"Copenhagen, Denmark");
```

To get the address for a latitude and longitude we can do:

```
from("direct:start")
.to("geocoder:latlng:40.714224,-73.961452")
.log("Location ${header.CamelGeocoderAddress} is at lat/lng:
${header.CamelGeocoderLatlng}
and in country ${header.CamelGeoCoderCountryShort}")
```

Which will log

```
Location 285 Bedford Avenue, Brooklyn, NY 11211, USA is at lat/lng:
40.71412890,-73.96140740 and in country US
```

To get the current location you can use "current" as the address as shown:

```
from("direct:start")
.to("geocoder:address:current")
```

3.27. Guava EventBus

The [Google Guava EventBus](#) allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other). The **guava-eventbus** component provides integration bridge between Camel and [Google Guava EventBus](#) infrastructure. With the latter component, messages exchanged with the Guava EventBus can be transparently forwarded to the Camel routes. EventBus component allows also to route body of Camel exchanges to the Guava EventBus.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-guava-eventbus</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.27.1. URI format

```
guava-eventbus:busName[?options]
```

Where **busName** represents the name of the `com.google.common.eventbus.EventBus` instance located in the Camel registry.

3.27.2. Options

Name	Default Value	Description
eventClass	null	Camel 2.10: If used on the consumer side of the route, will filter events received from the EventBus to the instances of the class and superclasses

Name	Default Value	Description
		of eventClass. Null value of this option is equal to setting it to the java.lang.Object i.e. the consumer will capture all messages incoming to the event bus. This option cannot be used together with listenerInterface option.
listenerInterface	null	Camel 2.11: The interface with method(s) marked with the @Subscribe annotation. Dynamic proxy will be created over the interface so it could be registered as the EventBus listener. Particularly useful when creating multi-event listeners and for handling DeadEvent properly. This option cannot be used together with eventClass option.

3.27.3. Usage

Using guava-eventbus component on the consumer side of the route will capture messages sent to the Guava EventBus and forward them to the Camel route. Guava EventBus consumer processes incoming messages [asynchronously](#).

```
SimpleRegistry registry = new SimpleRegistry();
EventBus eventBus = new EventBus();
registry.put("busName", eventBus);
CamelContext camel = new DefaultCamelContext(registry);

from("guava-eventbus:busName").to("seda:queue");

eventBus.post("Send me to the SEDA queue.");
```

Using guava-eventbus component on the producer side of the route will forward body of the Camel exchanges to the Guava EventBus instance.

```
SimpleRegistry registry = new SimpleRegistry();
EventBus eventBus = new EventBus();
registry.put("busName", eventBus);
CamelContext camel = new DefaultCamelContext(registry);

from("direct:start").to("guava-eventbus:busName");

ProducerTemplate producerTemplate = camel.createProducerTemplate();
producer.sendBody("direct:start", "Send me to the Guava EventBus.");

eventBus.register(new Object(){
    @Subscribe
    public void messageHandler(String message) {
        System.out.println("Message received from the Camel: " + message);
    }
});
```

3.27.4. DeadEvent considerations

Keep in mind that due to the limitations caused by the design of the Guava EventBus, you cannot specify event class to be received by the listener without creating class annotated with @Subscribe method. This limitation implies that endpoint with eventClass option specified actually listens to all possible events (java.lang.Object) and filter appropriate messages programmatically at runtime. The snippet below demonstrates an appropriate excerpt from the Camel code base.

```
@Subscribe
public void eventReceived(Object event) {
    if (eventClass == null || eventClass.isAssignableFrom(event.getClass())) {
```

```
doEventReceived(event);
...
```

This drawback of this approach is that `EventBus` instance used by Camel will never generate `com.google.common.eventbus.DeadEvent` notifications. If you want Camel to listen only to the precisely specified event (and therefore enable `DeadEvent` support), use `listenerInterface` endpoint option. Camel will create dynamic proxy over the interface you specify with the latter option and listen only to messages specified by the interface handler methods. The example of the listener interface with single method handling only `SpecificEvent` instances is demonstrated below.

```
package com.example;

public interface CustomListener {

    @Subscribe
    void eventReceived(SpecificEvent event);

}
```

The listener presented above could be used in the endpoint definition as follows.

```
from("guava-eventbus:busName?listenerInterface=com.example.CustomListener")
    .to("seda:queue");
```

3.27.5. Consuming multiple type of events

In order to define multiple type of events to be consumed by Guava `EventBus` consumer use `listenerInterface` endpoint option, as listener interface could provide multiple methods marked with the `@Subscribe` annotation.

```
package com.example;

public interface MultipleEventsListener {

    @Subscribe
    void someEventReceived(SomeEvent event);

    @Subscribe
    void anotherEventReceived(AnotherEvent event);

}
```

The listener presented above could be used in the endpoint definition as follows.

```
from("guava-eventbus:busName?listenerInterface=com.example.MultipleEventsListener")
    .to("seda:queue");
```

3.28. HBase

This component provides an idempotent repository, producers and consumers for [Apache HBase](#).

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hbase</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
```

```
</dependency>
```

3.28.1. Apache HBase Overview

HBase is an open-source, distributed, versioned, column-oriented store modeled after Google's Bigtable: A Distributed Storage System for Structured Data. You can use HBase when you need random, realtime read/write access to your Big Data. More information at [Apache HBase](#).

3.28.2. Camel and HBase

When using a datastore inside a camel route, there is always the challenge of specifying how the camel message will be stored to the datastore. In document based stores things are more easy as the message body can be directly mapped to a document. In relational databases an ORM solution can be used to map properties to columns etc. In column based stores things are more challenging as there is no standard way to perform that kind of mapping.

HBase adds two additional challenges:

- HBase groups columns into families, so just mapping a property to a column using a name convention is just not enough.
- HBase doesn't have the notion of type, which means that it stores everything as `byte[]` and doesn't know if the `byte[]` represents a String, a Number, a serialized Java object or just binary data.

To overcome these challenges, camel-hbase makes use of the message headers to specify the mapping of the message to HBase columns. It also provides the ability to use some camel-hbase provided classes that model HBase data and can be easily converted to and from xml/json etc.

Finally it provides the ability to the user to implement and use his own mapping strategy.

Regardless of the mapping strategy camel-hbase will convert a message into an `org.apache.camel.component.hbase.model.HBaseData` object and use that object for its internal operations.

3.28.3. Configuring the component

The HBase component can be provided a custom `HBaseConfiguration` object as a property or it can create an HBase configuration object on its own based on the HBase related resources that are found on classpath.

```
<bean id="hbase" class="org.apache.camel.component.hbase.HBaseComponent">
  <property name="configuration" ref="config"/>
</bean>
```

If no configuration object is provided to the component, the component will create one. The created configuration will search the class path for an `hbase-site.xml` file, from which it will draw the configuration. You can find more information about how to configure HBase clients at: [HBase client configuration and dependencies](#)

3.28.4. HBase Producer

As mentioned above camel provides producers endpoints for HBase. This allows you to store, delete, retrieve or query data from HBase using your camel routes.

```
hbase://table[?options]
```

where **table** is the table name.

The supported operations are:

- Put
- Get
- Delete
- Scan

3.28.4.1. Supported URI options on producer

Name	Default Value	Description
operation	CamelHBasePut	The HBase operation to perform. Supported values: CamelHBasePut, CamelHBaseGet, CamelHBaseDelete, and CamelHBaseScan.
maxResults	100	The maximum number of rows to scan. Supported operations: CamelHBaseScan.
mappingStrategyName	header	The strategy to use for mapping Camel messages to HBase columns. Supported values: header, or body.
mappingStrategyClassName	null	The class name of a custom mapping strategy implementation.
filters	null	A list of filters. Supported operations: CamelHBaseScan.

Header mapping options:

Name	Default Value	Description
rowId		The id of the row. This has limited use as the row usually changes per Exchange.
rowType	String	The type to covert row id to. Supported operations: CamelHBaseScan.
family		The column family. Supports a number suffix for referring to more than one columns
qualifier		The column qualifier. Supports a number suffix for referring to more than one columns
value		The value. Supports a number suffix for referring to more than one columns
valueType	String	The value type. Supports a number suffix for referring to more than one columns. Supported operations: CamelHBaseGet, and CamelHBaseScan.

3.28.4.2. Put Operations.

HBase is a column based store, which allows you to store data into a specific column of a specific row. Columns are grouped into families, so in order to specify a column you need to specify the column family and the qualifier of that column. To store data into a specific column you need to specify both the column and the row.

The simplest scenario for storing data into HBase from a camel route, would be to store part of the message body to specified HBase column.

```
<route>
  <from uri="direct:in"/>
  <!-- Set the HBase Row -->
```

```

<setHeader headerName="CamelHBaseRowId">
  <el>${in.body.id}</el>
</setHeader>
<!-- Set the HBase Value -->
<setHeader headerName="CamelHBaseValue">
  <el>${in.body.value}</el>
</setHeader>
<to
  uri="hbase:mytable?opertaion=CamelHBasePut&family=myfamily&qualifier=
  myqualifier"/>
</route>

```

The route above assumes that the message body contains an object that has an id and value property and will store the content of value in the HBase column myfamily:myqualifier in the row specified by id. If we needed to specify more than one column/value pairs we could just specify additional column mappings. Notice that you must use numbers from the 2nd header onwards, eg RowId2, RowId3, RowId4, etc. Only the 1st header does not have the number 1.

```

<route>
  <from uri="direct:in"/>
  <!-- Set the HBase Row 1st column -->
  <setHeader headerName="CamelHBaseRowId">
    <el>${in.body.id}</el>
  </setHeader>
  <!-- Set the HBase Row 2nd column -->
  <setHeader headerName="CamelHBaseRowId2">
    <el>${in.body.id}</el>
  </setHeader>
  <!-- Set the HBase Value for 1st column -->
  <setHeader headerName="CamelHBaseValue">
    <el>${in.body.value}</el>
  </setHeader>
  <!-- Set the HBase Value for 2nd column -->
  <setHeader headerName="CamelHBaseValue2">
    <el>${in.body.othervalue}</el>
  </setHeader>
  <to
    uri="hbase:mytable?opertaion=CamelHBasePut&family=myfamily&qualifier=
    myqualifier&family2=myfamily&qualifier2=myqualifier2"/>
  </route>

```

It is important to remember that you can use uri options, message headers or a combination of both. It is recommended to specify constants as part of the uri and dynamic values as headers. If something is defined both as header and as part of the uri, the header will be used.

3.28.4.3. Get Operations.

A Get Operation is an operation that is used to retrieve one or more values from a specified HBase row. To specify what are the values that you want to retrieve you can just specify them as part of the uri or as message headers.

```

<route>
  <from uri="direct:in"/>
  <!-- Set the HBase Row of the Get -->
  <setHeader headerName="CamelHBaseRowId">
    <el>${in.body.id}</el>
  </setHeader>
  <to
    uri="hbase:mytable?opertaion=CamelHBaseGet&family=myfamily&qualifier=
    myqualifier&valueType=java.lang.Long"/>
    <to uri="log:out"/>
  </route>

```

In the example above the result of the get operation will be stored as a header with name CamelHBaseValue.

3.28.4.4. Delete Operations.

You can also use camel-hbase to perform HBase delete operation. The delete operation will remove an entire row. All that needs to be specified is one or more rows as part of the message headers.

```
<route>
  <from uri="direct:in"/>
  <!-- Set the HBase Row of the Get -->
  <setHeader headerName="CamelHBaseRowId">
    <el>${in.body.id}</el>
  </setHeader>
  <to uri="hbase:mytable?operation=CamelHBaseDelete"/>
</route>
```

3.28.4.5. Scan Operations.

A scan operation is the equivalent of a query in HBase. You can use the scan operation to retrieve multiple rows. To specify what columns should be part of the result and also specify how the values will be converted to objects you can use either uri options or headers.

```
<route>
  <from uri="direct:in"/>
  <to uri="hbase:mytable?operation=CamelHBaseScan&family=myfamily&qualifier=myqualifier&valueType=java.lang.Long&rowType=java.lang.String"/>
  <to uri="log:out"/>
</route>
```

In this case it's probable that you also need to specify a list of filters for limiting the results. You can specify a list of filters as part of the uri and camel will return only the rows that satisfy **ALL** the filters.

To have a filter that will be aware of the information that is part of the message, camel defines the `ModelAwareFilter`. This will allow your filter to take into consideration the model that is defined by the message and the mapping strategy.

When using a `ModelAwareFilter` camel-hbase will apply the selected mapping strategy to the in message, will create an object that models the mapping and will pass that object to the Filter.

For example to perform scan using as criteria the message headers, you can make use of the `ModelAwareColumnMatchingFilter` as shown below.

```
<route>
  <from uri="direct:scan"/>
  <!-- Set the Criteria -->
  <setHeader headerName="CamelHBaseFamily">
    <constant>name</constant>
  </setHeader>
  <setHeader headerName="CamelHBaseQualifier">
    <constant>first</constant>
  </setHeader>
  <setHeader headerName="CamelHBaseValue">
    <el>in.body.firstName</el>
  </setHeader>
  <setHeader headerName="CamelHBaseFamily2">
    <constant>name</constant>
  </setHeader>
  <setHeader headerName="CamelHBaseQualifier2">
    <constant>last</constant>
  </setHeader>
  <setHeader headerName="CamelHBaseValue2">
    <el>in.body.lastName</el>
  </setHeader>
```

```

<!-- Set additional fields that you want to be return by skipping value -->
<setHeader headerName="CamelHBaseFamily3">
  <constant>address</constant>
</setHeader>
<setHeader headerName="CamelHBaseQualifier3">
  <constant>country</constant>
</setHeader>
<to uri="hbase:mytable?opertaion=CamelHBaseScan&filters=#myFilterList"/>
</route>

<bean id="myFilters" class="java.util.ArrayList">
  <constructor-arg>
    <list>
      <bean class="org.apache.camel.component.hbase.filters.
        ModelAwareColumnMatchingFilter"/>
    </list>
  </constructor-arg>
</bean>

```

The route above assumes that a pojo is with properties firstName and lastName is passed as the message body, it takes those properties and adds them as part of the message headers. The default mapping strategy will create a model object that will map the headers to HBase columns and will pass that model the the ModelAwareColumnMatchingFilter. The filter will filter out any rows, that do not contain columns that match the model. It is like query by example.

3.28.5. HBase Consumer

The Camel HBase Consumer, will perform repeated scan on the specified HBase table and will return the scan results as part of the message. You can either specify header mapping (default) or body mapping. The later will just add the org.apache.camel.component.hbase.model.HBaseData as part of the message body.

```
hbase://table[?options]
```

You can specify the columns that you want to be return and their types as part of the uri options:

```
hbase:mutable?family=name&qualifer=first&valueType=java.lang.String&family=address
&qualifer=number&valueType2=java.lang.Integer&rowType=java.lang.Long
```

The example above will create a model object that is consisted of the specified fields and the scan results will populate the model object with values. Finally the mapping strategy will be used to map this model to the camel message.

3.28.5.1. Supported URI options on consumer

Name	Default Value	Description
initialDelay	1000	Milliseconds before the first polling starts.
delay	500	Milliseconds before the next poll.
useFixedDelay	true	Controls if fixed delay or fixed rate is used. See ScheduledExecutorService in JDK for details.
timeUnit	TimeUnit.MILLISECONDS	time unit for initialDelay and delay options.
runLoggingLevel	TRACE	Camel 2.8: The consumer logs a start/complete log line when it polls. This option allows you to configure the logging level for that.

Name	Default Value	Description
operation	CamelHBasePut	The HBase operation to perform. Supported values: CamelHBasePut, CamelHBaseGet, CamelHBaseDelete, and CamelHBaseScan.
maxResults	100	The maximum number of rows to scan. Supported operations: CamelHBaseScan.
mappingStrategyName	header	The strategy to use for mapping Camel messages to HBase columns. Supported values: header, or body.
mappingStrategyClassName	null	The class name of a custom mapping strategy implementation.
filters	null	A list of filters. Supported operations: CamelHBaseScan
remove	true	If the option is true, Camel HBase Consumer will remove the rows which it processes.

Header mapping options:

Name	Default Value	Description
rowId		The id of the row. This has limited use as the row usually changes per Exchange.
rowType	String	The type to covert row id to. Supported operations: CamelHBaseScan
family		The column family. *upports a number suffix for referring to more than one columns
qualifier		The column qualifier. *Supports a number suffix for referring to more than one columns
value		The value. Supports a number suffix for referring to more than one columns
rowModel	String	An instance of org.apache.camel.component.hbase.model.HBaseRow which describes how each row should be modeled

If the role of the rowModel is not clear, it allows you to construct the HBaseRow model programmatically instead of "describing" it with uri options (such as family, qualifier, type etc).

3.28.6. HBase Idempotent repository

The camel-hbase component also provides an idempotent repository which can be used when you want to make sure that each message is processed only once. The HBase idempotent repository is configured with a table, a column family and a column qualifier and will create to that table a row per message.

```
HBaseConfiguration configuration = HBaseConfiguration.create();
HBaseIdempotentRepository repository = new HBaseIdempotentRepository(configuration,
    tableName, family, qualifier);

from("direct:in")
    .idempotentConsumer(header("messageId"), repository)
    .to("log:out");
```

3.28.7. HBase Mapping

It was mentioned above that you the default mapping strategies are **header** and **body** mapping.

Below you can find some detailed examples of how each mapping strategy works.

3.28.7.1. HBase Header mapping Examples

The header mapping is the default mapping.

To put the value "myvalue" into HBase row "myrow" and column "myfamily:mycolumn" the message should contain the following headers:

Header	Value
CamelHBaseRowId	myrow
CamelHBaseFamily	myfamily
CamelHBaseQualifier	myqualifier
CamelHBaseValue	myvalue

To put more values for different columns and / or different rows you can specify additional headers suffixed with the index of the headers, e.g:

Header	Value
CamelHBaseRowId	myrow
CamelHBaseFamily	myfamily
CamelHBaseQualifier	myqualifier
CamelHBaseValue	myvalue
CamelHBaseRowId2	myrow2
CamelHBaseFamily2	myfamily
CamelHBaseQualifier2	myqualifier
CamelHBaseValue2	myvalue2

In the case of retrieval operations such as get or scan you can also specify for each column the type that you want the data to be converted to. For example:

Header	Value
CamelHBaseFamily	myfamily
CamelHBaseQualifier	myqualifier
CamelHBaseValueType	Long

Please note that in order to avoid boilerplate headers that are considered constant for all messages, you can also specify them as part of the endpoint uri, as you will see below.

3.28.7.2. Body mapping Examples

In order to use the body mapping strategy you will have to specify the option `mappingStrategy` as part of the uri, for example:

```
hbase:mytable?mappingStrategy=body
```

To use the body mapping strategy the body needs to contain an instance of `org.apache.camel.component.hbase.model.HBaseData`. You can construct:

```
HBaseData data = new HBaseData();
HBaseRow row = new HBaseRow();
row.setId("myRowId");
HBaseCell cell = new HBaseCell();
cell.setFamily("myfamily");
cell.setQualifier("myqualifier");
cell.setValue("myValue");
row.getCells().add(cell);
```

```
data.addRows().add(row);
```

The object above can be used for example in a put operation and will result in creating or updating the row with id myRowId and add the value myvalue to the column myfamily:myqualifier.

The body mapping strategy might not seem very appealing at first. The advantage it has over the header mapping strategy is that the HBaseData object can be easily converted to or from xml/json.

3.29. HDFS

The **hdfs** component enables you to read and write messages from/to an HDFS file system. HDFS is the distributed file system at the heart of [Hadoop](#).

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hdfs</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.29.1. URI format

```
hdfs://hostname[:port][[/path]][?options]
```

You can append query options to the URI in the following format, ?option=value&option=value&...

The path is treated in the following way:

1. as a consumer, if it's a file, it just reads the file, otherwise if it represents a directory it scans all the file under the path satisfying the configured pattern. All the files under that directory must be of the same type.
2. as a producer, if at least one split strategy is defined, the path is considered a directory and under that directory the producer creates a different file per split named using the configured [UuidGenerator](#).

3.29.2. Options

Name	Default Value	Description
overwrite	true	The file can be overwritten
append	false	Append to existing file. Notice that not all HDFS file systems support the append option.
bufferSize	4096	The buffer size used by HDFS
replication	3	The HDFS replication factor
blocksize	67108864	The size of the HDFS blocks
fileType	NORMAL_FILE	It can be SEQUENCE_FILE, MAP_FILE, ARRAY_FILE, or BLOOMMAP_FILE, see Hadoop
fileSystemType	HDFS	It can be LOCAL for local filesystem
keyType	NULL	The type for the key in case of sequence or map files. See below.
valueType	TEXT	The type for the key in case of sequence or map files. See below.

Name	Default Value	Description
splitStrategy		A string describing the strategy on how to split the file based on different criteria. See below.
openedSuffix	opened	When a file is opened for reading/writing the file is renamed with this suffix to avoid to read it during the writing phase.
readSuffix	read	Once the file has been read is renamed with this suffix to avoid to read it again.
initialDelay	0	For the consumer, how much to wait (milliseconds) before to start scanning the directory.
delay	0	The interval (milliseconds) between the directory scans.
pattern	*	The pattern used for scanning the directory
chunkSize	4096	When reading a normal file, this is split into chunks producing a message per chunk.
connectOnStartup	true	Camel 2.9.3/2.10.1: Whether to connect to the HDFS file system on starting the producer/consumer. If <code>false</code> then the connection is created on-demand. Notice that HDFS may take up till 15 minutes to establish a connection, as it has hardcoded 45 x 20 sec redelivery. By setting this option to <code>false</code> allows your application to startup, and not block for up till 15 minutes.
owner		Camel 2.13/2.12.4: The file owner must match this owner for the consumer to pickup the file. Otherwise the file is skipped.

3.29.2.1. KeyType and ValueType

- NULL it means that the key or the value is absent
- BYTE for writing a byte, the java Byte class is mapped into a BYTE
- BYTES for writing a sequence of bytes. It maps the java ByteBuffer class
- INT for writing java integer
- FLOAT for writing java float
- LONG for writing java long
- DOUBLE for writing java double
- TEXT for writing java strings

BYTES is also used with everything else, for example, in Camel a file is sent around as an `InputStream`, in this case is written in a sequence file or a map file as a sequence of bytes.

3.29.3. Splitting Strategy

In the current version of Hadoop opening a file in append mode is disabled since it's not very reliable. So, for the moment, it's only possible to create new files. The Camel HDFS endpoint tries to solve this problem in this way:

- If the split strategy option has been defined, the hdfs path will be used as a directory and files will be created using the configured [UuidGenerator](#)
- Every time a splitting condition is met, a new file is created.

The `splitStrategy` option is defined as a string with the following syntax:

```
splitStrategy=<ST>:<value>,<ST>:<value>,*
```

where <ST> can be:

- BYTES a new file is created, and the old is closed when the number of written bytes is more than <value>
- MESSAGES a new file is created, and the old is closed when the number of written messages is more than <value>
- IDLE a new file is created, and the old is closed when no writing happened in the last <value> milliseconds

Note that this strategy currently requires either setting an IDLE value or setting the `HdfsConstants.HDFS_CLOSE` header to false to use the BYTES/MESSAGES configuration...otherwise, the file will be closed with each message

for example:

```
hdfs://localhost/tmp/simple-file?splitStrategy=IDLE:1000,BYTES:5
```

it means: a new file is created either when it has been idle for more than 1 second or if more than 5 bytes have been written. So, running `hadoop fs -ls /tmp/simple-file` you'll see that multiple files have been created.

3.29.4. Message Headers

The following headers are supported by this component:

3.29.4.1. Producer only

Header	Description
<code>CamelFileName</code>	Camel 2.13: Specifies the name of the file to write (relative to the endpoint path). The name can be a String or an Expression object. Only relevant when not using a split strategy.

3.29.5. Controlling to close file stream

Available as of Camel 2.10.4

When using the [HDFS](#) producer **without** a split strategy, then the file output stream is by default closed after the write. However you may want to keep the stream open, and only explicitly close the stream later. For that you can use the header `HdfsConstants.HDFS_CLOSE` (value = `"CamelHdfsClose"`) to control this. Setting this value to a boolean allows you to explicit control whether the stream should be closed or not.

Notice this does not apply if you use a split strategy, as there are various strategies that can control when the stream is closed.

3.29.6. Using this component in OSGi

This component is fully functional in an OSGi environment, however, it requires some actions from the user. Hadoop uses the thread context class loader in order to load resources. Usually, the thread context classloader will be the bundle class loader of the bundle that contains the routes. So, the default configuration files need to be visible from the bundle class loader. A typical way to deal with it is to keep a copy of `core-default.xml` in your bundle root. That file can be found in the `hadoop-common.jar`.

3.30. HDFS2

The **hdfs2** component enables you to read and write messages from/to an HDFS file system using Hadoop 2.x. HDFS is the distributed file system at the heart of [Hadoop](#).

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hdfs2</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.30.1. URI format

```
hdfs2://hostname[:port][/path][?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

The path is treated in the following way:

1. as a consumer, if it's a file, it just reads the file, otherwise if it represents a directory it scans all the file under the path satisfying the configured pattern. All the files under that directory must be of the same type.
2. as a producer, if at least one split strategy is defined, the path is considered a directory and under that directory the producer creates a different file per split named using the configured [UuidGenerator](#).

3.30.2. Options

Name	Default Value	Description
overwrite	true	The file can be overwritten
append	false	Append to existing file. Notice that not all HDFS file systems support the append option.
bufferSize	4096	The buffer size used by HDFS
replication	3	The HDFS replication factor
blocksize	67108864	The size of the HDFS blocks
fileType	NORMAL_FILE	It can be SEQUENCE_FILE, MAP_FILE, ARRAY_FILE, or BLOOMMAP_FILE, see Hadoop
fileSystemType	HDFS	It can be LOCAL for local filesystem
keyType	NULL	The type for the key in case of sequence or map files. See below.
valueType	TEXT	The type for the key in case of sequence or map files. See below.
splitStrategy		A string describing the strategy on how to split the file based on different criteria. See below.
openedSuffix	opened	When a file is opened for reading/writing the file is renamed with this suffix to avoid to read it during the writing phase.
readSuffix	read	Once the file has been read is renamed with this suffix to avoid to read it again.
initialDelay	0	For the consumer, how much to wait (milliseconds) before to start scanning the directory.
delay	0	The interval (milliseconds) between the directory scans.

Name	Default Value	Description
pattern	*	The pattern used for scanning the directory
chunkSize	4096	When reading a normal file, this is split into chunks producing a message per chunk.
connectOnStartup	true	Camel 2.9.3/2.10.1: Whether to connect to the HDFS file system on starting the producer/consumer. If <code>false</code> then the connection is created on-demand. Notice that HDFS may take up till 15 minutes to establish a connection, as it has hardcoded 45 x 20 sec redelivery. By setting this option to <code>false</code> allows your application to startup, and not block for up till 15 minutes.
owner		Camel 2.13/2.12.4: The file owner must match this owner for the consumer to pickup the file. Otherwise the file is skipped.

3.30.2.1. KeyType and ValueType

- NULL it means that the key or the value is absent
- BYTE for writing a byte, the java Byte class is mapped into a BYTE
- BYTES for writing a sequence of bytes. It maps the java ByteBuffer class
- INT for writing java integer
- FLOAT for writing java float
- LONG for writing java long
- DOUBLE for writing java double
- TEXT for writing java strings

BYTES is also used with everything else, for example, in Camel a file is sent around as an `InputStream`, in this case is written in a sequence file or a map file as a sequence of bytes.

3.30.3. Splitting Strategy

In the current version of Hadoop opening a file in append mode is disabled since it's not very reliable. So, for the moment, it's only possible to create new files. The Camel HDFS endpoint tries to solve this problem in this way:

- If the split strategy option has been defined, the hdfs path will be used as a directory and files will be created using the configured [UuidGenerator](#)
- Every time a splitting condition is met, a new file is created.

The `splitStrategy` option is defined as a string with the following syntax:

```
splitStrategy=<ST>:<value>,<ST>:<value>,*
```

where `<ST>` can be:

- BYTES a new file is created, and the old is closed when the number of written bytes is more than `<value>`
- MESSAGES a new file is created, and the old is closed when the number of written messages is more than `<value>`
- IDLE a new file is created, and the old is closed when no writing happened in the last `<value>` milliseconds

Note that this strategy currently requires either setting an IDLE value or setting the `HdfsConstants.HDFS_CLOSE` header to false to use the BYTES/MESSAGES configuration...otherwise, the file will be closed with each message

for example:

```
hdfs2://localhost/tmp/simple-file?splitStrategy=IDLE:1000,BYTES:5
```

it means: a new file is created either when it has been idle for more than 1 second or if more than 5 bytes have been written. So, running `hadoop fs -ls /tmp/simple-file` you'll see that multiple files have been created.

3.30.4. Message Headers

The following headers are supported by this component:

3.30.4.1. Producer only

Header	Description
<code>CamelFileName</code>	Camel 2.13: Specifies the name of the file to write (relative to the endpoint path). The name can be a String or an Expression object. Only relevant when not using a split strategy.

3.30.5. Controlling to close file stream

When using the [HDFS2](#) producer **without** a split strategy, then the file output stream is by default closed after the write. However you may want to keep the stream open, and only explicitly close the stream later. For that you can use the header `HdfsConstants.HDFS_CLOSE` (value = `"CamelHdfsClose"`) to control this. Setting this value to a boolean allows you to explicit control whether the stream should be closed or not.

Notice this does not apply if you use a split strategy, as there are various strategies that can control when the stream is closed.

3.30.6. Using this component in OSGi

There are some quirks when running this component in an OSGi environment related to the mechanism Hadoop 2.x uses to discover different `org.apache.hadoop.fs.FileSystem` implementations. Hadoop 2.x uses `java.util.ServiceLoader` which looks for `/META-INF/services/org.apache.hadoop.fs.FileSystem` files defining available filesystem types and implementations. These resources are not available when running inside OSGi.

As with `camel-hdfs` component, the default configuration files need to be visible from the bundle class loader. A typical way to deal with it is to keep a copy of `core-default.xml` (and e.g., `hdfs-default.xml`) in your bundle root.

3.30.6.1. Using this component with manually defined routes

There are two options:

1. Package `/META-INF/services/org.apache.hadoop.fs.FileSystem` resource with bundle that defines the routes. This resource should list all the required Hadoop 2.x filesystem implementations.
2. Provide boilerplate initialization code which populates internal, static cache inside `org.apache.hadoop.fs.FileSystem` class:

```
org.apache.hadoop.conf.Configuration conf = new
    org.apache.hadoop.conf.Configuration();
conf.setClass("fs.file.impl", org.apache.hadoop.fs.LocalFileSystem.class,
    FileSystem.class);
conf.setClass("fs.hdfs.impl", org.apache.hadoop.hdfs.DistributedFileSystem.class,
    FileSystem.class);
...
FileSystem.get("file://", conf);
FileSystem.get("hdfs://localhost:9000/", conf);
...
```

3.30.6.2. Using this component with Blueprint container

Two options:

1. Package `/META-INF/services/org.apache.hadoop.fs.FileSystem` resource with bundle that contains blueprint definition.
2. Add the following to the blueprint definition file:

```
<bean id="hdfsOsgiHelper" class="org.apache.camel.component.hdfs2.HdfsOsgiHelper">
    <argument>
        <map>
            <entry key="file://" value="org.apache.hadoop.fs.LocalFileSystem" />
            <entry key="hdfs://localhost:9000/"
                value="org.apache.hadoop.hdfs.DistributedFileSystem" />
            ...
        </map>
    </argument>
</bean>

<bean id="hdfs2" class="org.apache.camel.component.hdfs2.HdfsComponent" depends-
on="hdfsOsgiHelper" />
```

This way Hadoop 2.x will have correct mapping of URI schemes to filesystem implementations.

3.31. HI7

The `hl7` component is used for working with the HL7 MLLP protocol and [HL7 v2 messages](#) 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 and Camel-Mina2 (for Camel 2.11+) components.

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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.31.1. 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 or Mina2 components 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 charset name encoding to use for the codec. If not provided, Camel will use the JVM default Charset .
convertLFtoCR	true (Camel 2.11 : false)	Will convert <code>\n</code> to <code>\r</code> (0x0d, 13 decimal) as HL7 usually uses <code>\r</code> as segment terminators. The HAPI library requires the use of <code>\r</code> . Default value of true pre-Camel 2.11, false starting with Camel 2.11.
validate	true	Whether HAPI Parser should validate or not.
parser	ca.uhn.hl7v2.parser.PipeParser	Starting with Camel 2.11, to use a custom parser. Must be of type <code>ca.uhn.hl7v2.parser.Parser</code> .

3.31.1.1. 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"/>
<!-- for Camel 2.11: use uri="mina2:tcp..." -->
```

Notice that we use TCP on 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("hl7listener").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.healthcare.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:

```
import ca.uhn.hl7v2.HL7Exception;
import ca.uhn.hl7v2.model.Message;
import ca.uhn.hl7v2.model.v24.segment.QRD;

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 imports from the HAPI library and not from Camel.

3.31.2. 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.

3.31.3. HL7 Model using HAPI

The HL7v2 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 HL7v2. 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 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` object. To retrieve the patient ID in the message 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);
```

If you know the message type in advance, you can be more type-safe:

```
QRY_A19 msg = exchange.getIn().getBody(QRY_A19.class);
String patientId = msg.getQRD().getWhoSubjectFilter(0).getIDNumber().getValue();
```

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 HL7v2 model objects.

3.31.4. 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

3.31.5. Options

The HL7 Data Format supports the following options:

Option	Default	Description
validate	true	Whether the HAPI Parser should validate using the default validation rules. Camel 2.11: better use the <code>{{parser}}</code> option and initialize the parser with the desired HAPI <code>{{ValidationContext}}</code>
parser	ca.uhn.hl7v2.parser.GenericParser	Starting with Camel 2.11, to use a custom parser. Must be of type <code>ca.uhn.hl7v2.parser.Parser</code> . Note that <code>GenericParser</code> also allows for parsing XML-encoded HL7v2 messages.

3.31.6. 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>
<!-- use the same version as your hapi-base version -->
<version>1.2</version>
</dependency>
```

```
<dependency>
  <groupId>ca.uhn.hapi</groupId>
  <artifactId>hapi-structures-v25</artifactId>
  <!-- use the same version as your hapi-base version -->
  <version>1.2</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 [central Maven repository](#):

```
<dependency>
  <groupId>ca.uhn.hapi</groupId>
  <artifactId>hapi-osgi-base</artifactId>
  <version>1.2</version>
</dependency>
```

Note that the version number must match the version of the hapi-base library that is transitively referenced by this component.

See the [Camel Website](#) for examples of this component in use.

3.31.7. Terse language (Camel 2.11)

HAPI provides a [Terse](#) class that provides access to fields using a commonly used terse location specification syntax. The Terse language allows to use this syntax to extract values from messages and to use them as expressions and predicates for filtering, content-based routing etc.

Sample:

```
import static org.apache.camel.component.hl7.HL7.terse;
...

// extract patient ID from field QRD-8 in the QRY_A19 message above and put into
message header
from("direct:test1")
  .setHeader("PATIENT_ID",terse("QRD-8(0)-1"))
  .to("mock:test1");
// continue processing if extracted field equals a message header
from("direct:test2")
  .filter(terse("QRD-8(0)-1"))
  .isEqualTo(header("PATIENT_ID"))
  .to("mock:test2");
```

3.31.8. HL7 Validation predicate (Camel 2.11)

Often it is preferable to parse a HL7v2 message and validate it against a **HAPI ValidationContext** in a separate step afterwards.

Sample:

```
import static org.apache.camel.component.hl7.HL7.messageConformsTo;
import ca.uhn.hl7v2.validation.impl.DefaultValidation;
```

```
...

// Use standard or define your own validation rules
ValidationContext defaultContext = new DefaultValidation();

// Throws PredicateValidationException if message does not validate
from("direct:test1").validate(messageConformsTo(defaultContext)).to("mock:test1");
```

3.31.9. HL7 Acknowledgement expression (Camel 2.11)

A common task in HL7v2 processing is to generate an acknowledgement message as response to an incoming HL7v2 message, e.g. based on a validation result. The ack expression lets us accomplish this very elegantly:

```
import static org.apache.camel.component.hl7.HL7.messageConformsTo;
import static org.apache.camel.component.hl7.HL7.ack;
import ca.uhn.hl7v2.validation.impl.DefaultValidation;
...

// Use standard or define your own validation rules
ValidationContext defaultContext = new DefaultValidation();

from("direct:test1")
    .onException(Exception.class)
    .handled(true)
    .transform(ack()) // auto-generates negative ack because of exception in
Exchange
    .end()
    .validate(messageConformsTo(defaultContext))
    // do something meaningful here
    ...
    // acknowledgement
    .transform(ack())
```

3.32. HTTP4

The **http4:** component provides HTTP based [endpoints](#) for calling 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.32.1. 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&...`



Should you use camel-http4 or 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.

3.32.2. HttpComponent Options

Table 3.10.

Name	Default Value	Description
maxTotalConnections	200	The maximum number of connections.
connectionsPerRoute	20	The maximum number of connections per route.
cookiestore	null	Camel 2.11.2/2.12.0: To use a custom org.apache. http.client.CookieStore. By default the org.apache.http.impl. client.BasicCookieStore is used which is an in-memory only cookie store. Notice if bridgeEndpoint=true then the cookie store is forced to be a noop cookie store as cookies shouldn't be stored as we are just bridging (eg acting as a proxy).
httpClientConfigurer	null	Reference to a org.apache.camel.component. http.HttpClientConfigurer in the Registry.
clientConnectionManager	null	To use a custom org.apache.http. conn.ClientConnectionManager.
httpBinding	null	To use a custom org.apache.camel. component.http.HttpBinding.
httpContext	null	To use a custom org.apache.http. protocol.HttpContext when executing requests.
sslContextParameters	null	To use a custom org.apache.camel.util. jsse.SSLContextParameters. See Using the JSSE Configuration Utility. Important: Only one instance of org.apache.camel.util. jsse.SSLContextParameters is supported per HttpComponent. If you need to use 2 or more different instances, you need to define a new HttpComponent per instance you need. See further below for more details.
x509HostnameVerifier	BrowserCompatHost nameVerifier	Camel 2.7 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.
connectionTimeToLive	-1	Camel 2.11.0: The time for connection to live, the time unit is millisecond, the default value is always keep alive.

Name	Default Value	Description
authenticationPreemptive	false	Camel 2.11.3/2.12.2: If this option is true, camel-http4 sends preemptive basic authentication to the server.

3.32.3. HttpEndpoint Options

Table 3.11.

Name	Default Value	Description
throwExceptionOnFailure	true	Option to disable throwing the <code>HttpOperationFailedException</code> 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, <code>HttpProducer</code> will ignore the <code>Exchange.HTTP_URI</code> header, and use the endpoint's URI for requests. You may also set the throwExceptionOnFailure to be false to let the <code>HttpProducer</code> send all the fault response back. Also if set to true <code>HttpProducer</code> and <code>CamelServlet</code> will skip the gzip processing if the content-encoding is "gzip".
clearExpiredCookies	true	Camel 2.11.2/2.12.0: Whether to clear expired cookies before sending the HTTP request. This ensures the cookies store does not keep growing by adding new cookies which is newer removed when they are expired.
cookieStore	null	Camel 2.11.2/2.12.0: To use a custom <code>org.apache.http.client.CookieStore</code> . By default the <code>org.apache.http.impl.client.BasicCookieStore</code> is used which is an in-memory only cookie store. Notice if <code>bridgeEndpoint=true</code> then the cookie store is forced to be a noop cookie store as cookies shouldn't be stored as we are just bridging (eg acting as a proxy).
disableStreamCache	false	<code>DefaultHttpBinding</code> 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 <code>DefaultHttpBinding</code> will set the request input stream directly in the message body.
headerFilterStrategy	null	Reference to a instance of <code>org.apache.camel.spi.HeaderFilterStrategy</code> in the Registry . It will be used to apply the custom headerFilterStrategy on the new create <code>HttpEndpoint</code> .
httpBindingRef	null	Deprecated and will be removed in Camel 3.0: Reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry . Use the <code>httpClientConfigurer</code> option instead.
httpBinding	null	To use a custom <code>org.apache.camel.component.http.HttpBinding</code> .
httpClientConfigurerRef	null	Deprecated and will be removed in Camel 3.0: Reference to a <code>org.apache.camel.component</code> .

Name	Default Value	Description
		<code>http.HttpClientConfigurer</code> in the Registry . Use the <code>httpClientConfigurer</code> option instead.
<code>httpClientConfigurer</code>	<code>null</code>	Reference to a <code>org.apache.camel.component.http.HttpClientConfigurer</code> in the Registry .
<code>httpContextRef</code>	<code>null</code>	Deprecated and will be removed in Camel 3.0:Camel 2.9.2: Reference to a custom <code>org.apache.http.protocol.HttpContext</code> in the Registry . Use the <code>httpContext</code> instead.
<code>httpContext</code>	<code>null</code>	Camel 2.9.2: To use a custom <code>org.apache.http.protocol.HttpContext</code> when executing requests.
<code>httpClient.XXX</code>	<code>null</code>	Setting options on the BasicHttpParams . For instance <code>httpClient.soTimeout=5000</code> will set the <code>SO_TIMEOUT</code> 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 Since Camel 2.13.0: <code>httpClient</code> is changed to configure the HttpClientBuilder and RequestConfig.Builder , please check out API document for a complete reference.
<code>clientConnectionManager</code>	<code>null</code>	To use a custom <code>org.apache.http.conn.ClientConnectionManager</code> .
<code>transferException</code>	<code>false</code>	If enabled and an Exchange failed processing on the consumer side, and if the caused <code>Exception</code> was send back serialized in the response as a <code>application/x-java-serialized-object</code> content type (for example using Jetty or Servlet Camel components). On the producer side the exception will be deserialized and thrown as is, instead of the <code>HttpOperationFailedException</code> . The caused exception is required to be serialized.
<code>sslContextParametersRef</code>	<code>null</code>	Deprecated and will be removed in Camel 3.0:Camel 2.8: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry . Important: Only one instance of <code>org.apache.camel.util.jsse.SSLContextParameters</code> is supported per <code>HttpComponent</code> . If you need to use 2 or more different instances, you need to define a new <code>HttpComponent</code> per instance you need. See further below for more details. See Using the JSSE Configuration Utility . Use the <code>sslContextParameters</code> option instead.
<code>sslContextParameters</code>	<code>null</code>	Camel 2.11.1: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry . Important: Only one instance of <code>org.apache.camel.util.jsse.SSLContextParameters</code> is supported per <code>HttpComponent</code> . If you need to use 2 or more different instances, you need to define a new <code>HttpComponent</code> per instance you need. See further below for more details. See Using the JSSE Configuration Utility .

Name	Default Value	Description
x509HostnameVerifier	BrowserCompatHost nameVerifier	Camel 2.7: You can refer to a different <code>org.apache.http.conn.ssl.X509HostnameVerifier</code> instance in the Registry such as <code>org.apache.http.conn.ssl.StrictHostnameVerifier</code> or <code>org.apache.http.conn.ssl.AllowAllHostnameVerifier</code> .
urlRewrite	null	Camel 2.11:Producer only Refers to a custom <code>org.apache.camel.component.http4.UrlRewrite</code> which allows you to rewrite urls when you bridge/proxy endpoints. See more details at UrlRewrite and How to use Camel as a HTTP proxy between a client and server .
maxTotalConnections	null	Camel 2.14: The maximum number of total connections that the connection manager has. If this option is not set, camel will use the component's setting instead.
connectionsPerRoute	null	Camel 2.14: The maximum number of connections per route. If this option is not set, camel will use the component's setting instead.

The following authentication options can also be set on the HttpEndpoint:

3.32.3.1. Setting Basic Authentication and Proxy

Before Camel 2.8.0

Name	Default Value	Description
authUsername	null	Username for authentication.
authPassword	null	Password for authentication.
authDomain	null	The domain name for authentication.
authHost	null	The host name authentication.
proxyAuthHost	null	The proxy host name
proxyAuthPort	null	The proxy port number
proxyAuthScheme	null	The proxy scheme, will fallback and use the scheme from the endpoint if not configured.
proxyAuthUsername	null	Username for proxy authentication
proxyAuthPassword	null	Password for proxy authentication
proxyAuthDomain	null	The proxy domain name
proxyAuthNtHost	null	The proxy Nt host name

Since Camel 2.8.0

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

Name	Default Value	Description
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

3.32.4. 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.
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.

3.32.5. 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.

3.32.6. 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.



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.

3.32.7. 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

3.32.8. 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.

3.32.9. 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);  
HttpServletResponse response =  
    exchange.getIn().getBody(HttpServletResponse.class);
```

3.32.10. Configuring URI to call

You can set the HTTP producer's URI directly from 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 `Exchange.HTTP_URI` on the message.

```
from("direct:start")  
    .setHeader(Exchange.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`. If the `http4` endpoint is working in bridge mode, it will ignore the `Exchange.HTTP_URI` message header.

3.32.11. 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");
```

3.32.12. How to set the http method (GET/POST/PUT/DELETE/HEAD/OPTIONS/TRACE) to the HTTP producer

Note: The http PATCH method is supported starting with Camel 2.11.3 / 2.12.1.

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>
```

3.32.13. 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.

3.32.13.1. 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.

3.32.14. 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");
```

3.32.14.1. 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");
```

3.32.14.2. 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);
```

3.32.14.3. URI Parameters from the Message

```
Map headers = new HashMap();
```



```
headers.put(Exchange.HTTP_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.

3.32.14.4. 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 `Exchange.HTTP_RESPONSE_CODE`.

```
Exchange exchange =
    template.send("http4://www.google.com/search", new Processor() {
        public void process(Exchange exchange) throws Exception {
            exchange.getIn().setHeader(
                Exchange.HTTP_QUERY, constant("hl=en&q=activemq"));
        }
    });
Message out = exchange.getOut();
int responseCode = out.getHeader(Exchange.HTTP_RESPONSE_CODE,
    Integer.class);
```

3.32.15. Disabling Cookies

To disable cookies you can set the HTTP Client to ignore cookies by adding this URI option:
`httpClient.cookiePolicy=ignoreCookies`

3.32.16. 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.

3.32.16.1. 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 designer 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.

3.33. Infinispan

Available as of Camel 2.13.0

This component allows you to interact with [Infinispan](#) distributed data grid / cache. Infinispan is an extremely scalable, highly available key/value data store and data grid platform written in Java.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-infinispan</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.33.1. URI format

```
infinispan://hostName?[options]
```

3.33.2. URI Options

The producer allows sending messages to a local infinispan cache configured in the registry, or to a remote cache using the HotRod protocol. The consumer allows listening for events from local infinispan cache accessible from the registry.

Name	Default Value	Type	Context	Description
cacheContainer	null	CacheContainer	Shared	Reference to a <code>org.infinispan.manager.CacheContainer</code> in the Registry .

Name	Default Value	Type	Context	Description
cacheName	null	String	Shared	The cache name to use. If not specified, default cache is used.
command	PUT	String	Producer	The operation to perform. Currently supports the following values: PUT, GET, REMOVE, CLEAR.
eventTypes	null	Set<String>	Consumer	The event types to register. By default will listen for all event types. Possible values defined in <code>org.infinispan.notifications</code> . <code>cachelistener.event.Event.Type</code>
sync	true	Boolean	Consumer	By default the consumer will receive notifications synchronously, by the same thread that process the cache operation.

3.33.3. Message Headers

Name	Default Value	Type	Context	Description
CamelInfinispanCacheName	null	String	Shared	The cache participating in the operation or event.
CamelInfinispanOperation	PUT	String	Producer	The operation to perform: <code>CamelInfinispanOperationPut</code> , <code>CamelInfinispanOperationGet</code> , <code>CamelInfinispanOperationRemove</code> , <code>CamelInfinispanOperationClear</code> .
CamelInfinispanKey	null	Object	Shared	The key to perform the operation to or the key generating the event.
CamelInfinispanValue	null	Object	Producer	The value to use for the operation.
CamelInfinispanOperationResult	null	Object	Producer	The result of the operation.
CamelInfinispanEventType	null	String	Consumer	The type of the received event. Possible values defined here <code>org.infinispan.notifications</code> . <code>cachelistener.event.Event.Type</code>
CamelInfinispanIsPre	null	Boolean	Consumer	Infinispan fires two events for each operation: one before and one after the operation.

3.33.4. Example

Below is an example route that retrieves a value from the cache for a specific key:

```
from("direct:start")
    .setHeader(InfinispanConstants.OPERATION, constant(InfinispanConstants.GET))
    .setHeader(InfinispanConstants.KEY, constant("123"))
    .to("infinispan://localhost?cacheContainer=#cacheContainer");
```

3.33.5. Using the Infinispan based idempotent repository

In this section we will use the Infinispan based idempotent repository.

First, we need to create a `cacheManager` and then configure our `org.apache.camel.component.infinispan.processor.idempotent`.

`InfinispanIdempotentRepository`:

```
<bean id="cacheManager" class="org.infinispan.manager.DefaultCacheManager"
init-method="start" destroy-method="stop" />
<bean id="infinispanRepo"
class="org.apache.camel.component.infinispan.processor.idempotent.
InfinispanIdempotentRepository"
factory-method="infinispanIdempotentRepository">
<argument ref="cacheManager"/>
<argument value="idempotent"/>
</bean>
```

Then we can create our `Infinispan idempotent repository` in the spring XML file as well:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
<route id="JpaMessageIdRepositoryTest">
<from uri="direct:start" />
<idempotentConsumer messageIdRepositoryRef="infinispanStore">
<header>messageId</header>
<to uri="mock:result" />
</idempotentConsumer>
</route>
</camelContext>
```

3.34. Jasypt

[Jasypt](#) is a simplified encryption library which makes encryption and decryption easy. Camel integrates with Jasypt to allow sensitive information in [Properties](#) files to be encrypted. By dropping `camel-jasypt` on the classpath those encrypted values will automatically 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>
<!-- use the same version as your Camel core version -->
<version>x.x.x</version>
</dependency>
```

Jasypt 1.7 onwards is fully standalone so no additional JARs are needed.

3.34.1. Tooling

The [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 [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==)
```

3.34.2. URI Options

The options below are exclusive for the [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.

3.34.3. Protecting the master password

The master password used by [Jasypt](#) must be provided, so that it's 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 provide 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`.

3.34.4. Example with Java DSL

In Java DSL you need to configure *Jasypt* as a `JasyptPropertiesParser` instance and set it on the *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==)
```

3.34.5. Example with Spring XML

In Spring XML you need to configure the `JasyptPropertiesParser` which is shown below. Then the Camel *Properties* component is told to use `jasypt` as the properties parser, which means *Jasypt* has 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 *Properties* component can also be inlined inside the `<camelContext>` tag which is shown below. Notice how we use the `propertiesParserRef` attribute to refer to *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 to 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>

```

3.35. JCR

The `jcr` component allows you to add nodes to a JCR (JSR-170) compliant content repository (for example, [Apache Jackrabbit](#)) using a producer, or listen for changes with a consumer.

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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>

```

3.35.1. URI format

```
jcr://user:password@repository/path/to/node
```

3.35.2. Usage

See the [Camel website](#) for the most up-to-date examples of this component in use.

The `repository` element of the URI is used to look up the `JCRRepository` object in the Camel context registry.

3.35.2.1. Producer

Name	Default Value	Description
<code>CamelJcrOperation</code>	<code>CamelJcrInsert</code>	<code>CamelJcrInsert</code> or <code>CamelJcrGetById</code> operation to use
<code>CamelJcrNodeName</code>	<code>null</code>	Used to determine the node name to use.

When a message is sent to a JCR producer endpoint:

- If the operation is `CamelJcrInsert`: A new node is created in the content repository, all the message headers of the IN message are transformed to `javax.jcr.Value` instances and added to the new node and the node's UUID is returned in the OUT message.
- If the operation is `CamelJcrGetById`: A new node is retrieved from the repository using the message body as node identifier.

Please note that the JCR Producer used message properties instead of message headers in Camel versions earlier than 2.12.3. See <https://issues.apache.org/jira/browse/CAMEL-7067> for more details.

3.35.2.2. Consumer

The consumer will connect to JCR periodically and return a `List<javax.jcr.observation.Event>` in the message body.

Name	Default Value	Description
<code>eventTypes</code>	<code>0</code>	A combination of one or more event types encoded as a bit mask value such as <code>javax.jcr.observation.Event.NODE_ADDED</code> , <code>javax.jcr.observation.Event.NODE_REMOVED</code> , etc.
<code>deep</code>	<code>false</code>	When it is true, events whose associated parent node is at current path or within its subgraph are received.
<code>uuids</code>	<code>null</code>	Only events whose associated parent node has one of the identifiers in the comma separated uuid list will be received.
<code>nodeTypeNames</code>	<code>null</code>	Only events whose associated parent node has one of the node types (or a subtype of one of the node types) in this list will be received.
<code>noLocal</code>	<code>false</code>	If <code>noLocal</code> is true, then events generated by the session through which the listener was registered are ignored. Otherwise, they are not ignored.
<code>sessionLiveCheckInterval</code>	<code>60000</code>	Interval in milliseconds to wait before each session live checking.
<code>sessionLiveCheckIntervalOnStart</code>	<code>3000</code>	Interval in milliseconds to wait before the first session live checking.

3.36. JDBC

The **jdb**c 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 [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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```



This component can only be used to define producer endpoints, which means that you cannot use the JDBC component in a `from()` statement.

3.36.1. 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&...`

3.36.2. Options

Name	Default Value	Description
readSize	0	The default maximum number of rows that can be read by a polling query.
statement.<xxx>	null	Sets additional options on the <code>java.sql.Statement</code> that is used behind the scenes to execute the queries. For instance, <code>statement.maxRows=10</code> . 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 <code>SELECT</code> using aliases (for example, SQL <code>SELECT id as identifier, name as given_name from persons</code>).
resetAutoCommit	true	Camel will set the <code>autoCommit</code> on the JDBC connection to be false, commit the change after executing the statement and reset the <code>autoCommit</code> flag of the connection at the end, if the <code>resetAutoCommit</code> is true. If the JDBC connection doesn't support resetting the <code>autoCommit</code> flag, you can set the <code>resetAutoCommit</code> flag to be false, and Camel will not try to reset the <code>autoCommit</code> flag.
allowNamedParameters	true	Camel 2.12: Whether to allow using named parameters in the queries.
prepareStatementStrategy		Camel 2.12: Allows to plugin to use a custom <code>org.apache.camel.component.jdbc.JdbcPrepareStatementStrategy</code> to control preparation of the query and prepared statement.

Name	Default Value	Description
useHeadersAsParameters	false	Camel 2.12: Set this option to true to use the <code>prepareStatementStrategy</code> with named parameters. This allows to define queries with named placeholders, and use headers with the dynamic values for the query placeholders.
outputType	SelectList	<p>Camel 2.12.1: Make the output of the producer to <code>SelectList</code> as List of Map, or <code>SelectOne</code> as single Java object in the following way:</p> <p>a) If the query has only single column, then that JDBC Column object is returned. (such as <code>SELECT COUNT(*) FROM PROJECT</code> will return a Long object.</p> <p>b) If the query has more than one column, then it will return a Map of that result.</p> <p>c) If the <code>outputClass</code> is set, then it will convert the query result into an Java bean object by calling all the setters that match the column names. It will assume your class has a default constructor to create instance with.</p> <p>d) If the query resulted in more than one rows, it throws a non-unique result exception.</p> <p>Camel 2.14.0: New <code>StreamList</code> output type value that streams the result of the query using an <code>Iterator<Map<String, Object>></code>, it can be used along with the Splitter EIP.</p>
outputClass	null	Camel 2.12.1: Specify the full package and class name to use as conversion when <code>outputType=SelectOne</code> .
beanRowMapper		<p>Camel 2.12.1: To use a custom <code>org.apache.camel.component.jdbc.BeanRowMapper</code> when using <code>outputClass</code>. The default implementation will lower case the row names and skip underscores, and dashes. For example <code>"CUST_ID"</code> is mapped as <code>"custId"</code>.</p>

3.36.3. Result

By default 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. You can use the option `outputType` to control the result.

Note: This component fetches `ResultSetMetaData` to be able to return the column name as the key in the `Map`.

3.36.3.1. Message Headers

Header	Description
CamelJdbcRowCount	If the query is a <code>SELECT</code> , query the row count is returned in this OUT header.
CamelJdbcUpdateCount	If the query is an <code>UPDATE</code> , query the update count is returned in this OUT header.
CamelGeneratedKeysRows	Rows that contain the generated keys. If you insert data using <code>SQL INSERT</code> , 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.
CamelJdbcColumnNames	The column names from the <code>ResultSet</code> as a <code>java.util.Set</code> type.
CamelJdbcParameters	A <code>java.util.Map</code> which has the headers to be used if <code>useHeadersAsParameters</code> has been enabled.

3.36.4. 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 [Splitter](#) EIP such as:

In Camel 2.13.x or older

```
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");
```

In Camel 2.14.x or newer

```
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
    // the StreamList option allows to stream the result of the query without creating a
    // List of rows
    // and notice we also enable streaming mode on the splitter
    .to("jdbc:testdb?outputType=StreamList")
    .split(body()).streaming()
    .to("mock:result");
```

3.37. 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.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jetty</artifactId>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

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 `Exchange.HTTP_RESPONSE_CODE` 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.

3.37.1. URI format

```
jetty:http://hostname[:port][/resourceUri][?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.37.2. Options

Name	Default Value	Description
<code>sessionSupport</code>	<code>false</code>	Specifies whether to enable the session manager on the server side of Jetty.
<code>httpClient.XXX</code>	<code>null</code>	Configuration of Jetty's HttpClient . For example, setting <code>httpClient.idleTimeout=30000</code> sets the idle timeout to 30 seconds.
<code>httpClient</code>	<code>null</code>	To use a shared HttpClient for all producers created by this endpoint. This option should only be used in special circumstances.
<code>httpClientMinThreads</code>	<code>null</code>	Starting with Camel 2.11: (Producer only): To set a value for minimum number of threads in the HttpClient thread pool. This setting overrides any setting configured at the component level. Notice that both a min and max size must be configured.
<code>httpClientMaxThreads</code>	<code>null</code>	Starting with Camel 2.11: (Producer only): To set a value for maximum number of threads in HttpClient thread pool. This setting overrides any setting configured at the component level. Notice that both a min and max size must be configured.
<code>httpBindingRef</code>	<code>null</code>	Reference to an Camel <code>HttpBinding</code> object in the Registry . <code>HttpBinding</code> can be used to customize how a response should be written for the consumer.
<code>jettyHttpBindingRef</code>	<code>null</code>	Reference to a Camel <code>JettyHttpBinding</code> object in the Registry . <code>JettyHttpBinding</code> can be used to customize how a response should be written for the producer.
<code>matchOnUriPrefix</code>	<code>false</code>	Whether or not the <code>CamelServlet</code> 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 .
<code>handlers</code>	<code>null</code>	Specifies a comma-delimited set of <code>org.mortbay.jetty.Handler</code> instances in your Registry (such as your <code>SpringApplicationContext</code>). These handlers are added to the Jetty servlet context (for example, to add security).
<code>chunked</code>	<code>true</code>	If this option is false Jetty servlet will disable the HTTP streaming and set the content-length header on the response
<code>enableJmx</code>	<code>false</code>	If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.
<code>disableStreamCache</code>	<code>false</code>	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 <code>true</code> when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. <code>DefaultHttpBinding</code> will copy the request input stream into a stream cache and put it into message body if this option is <code>false</code> 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.
<code>throwExceptionOnFailure</code>	<code>true</code>	Option to disable throwing the <code>HttpOperationFailedException</code> in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.
<code>transferException</code>	<code>false</code>	Camel 2.6: 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 <code>application/x-java-serialized-object</code> content type. On the producer side the exception will be deserialized and thrown as is, instead of the <code>HttpOperationFailedException</code> . The caused exception is required to be serialized.

Name	Default Value	Description
bridgeEndpoint	false	If the option is true, <code>HttpProducer</code> will ignore the <code>Exchange.HTTP_URI</code> header, and use the endpoint's URI for request. You may also set the throwExceptionOnFailure to be false to let the <code>HttpProducer</code> send all the fault response back. If the option is true, <code>HttpProducer</code> and <code>CamelServlet</code> will skip the gzip processing if the content-encoding is "gzip". Also consider setting <code>*disableStreamCache*</code> to true to optimize when bridging.
enableMultipartFilter	true	Whether <code>Jetty org.eclipse.jetty.servlets.MultipartFilter</code> 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	Allows using a custom multipart filter. Note: setting <code>multipartFilterRef</code> forces the value of <code>enableMultipartFilter</code> to true.
filtersRef	null	Allows using a custom filter which is put into a list and can be found in the Registry
continuationTimeout	null	Allows to set a timeout in milliseconds when using Jetty as consumer (server). By default <code>Jetty</code> uses 30000. You can use a value of <code><= 0</code> to never expire. If a timeout occurs then the request will be expired and <code>Jetty</code> will return back a http error 503 to the client. This option is only in use when using Jetty with the Asynchronous Routing Engine .
useContinuation	true	Whether or not to use Jetty continuations for the <code>Jetty Server</code> .
sslContextParametersRef	null	Reference to an <code>org.apache.camel.util.jsse.SSLContextParameters</code> object in the Camel Registry. This reference overrides any configured <code>SSLContextParameters</code> at the component level.
traceEnabled	false	Specifies whether to enable HTTP TRACE for this <code>Jetty</code> consumer. By default TRACE is turned off.
headerFilterStrategy	null	Starting with Camel 2.11: Reference to a instance of HeaderFilterStrategy in the Registry. It will be used to apply the custom <code>headerFilterStrategy</code> on the new create <code>HttpJettyEndpoint</code> .
urlRewrite	null	*Camel 2.11:* *Producer only* Refers to a custom UrlRewrite which allows you to rewrite urls when you bridge/proxy endpoints.
responseBufferSize	null	Camel 2.12: To use a custom buffer size on the <code>javax.servlet.ServletResponse</code> .
proxyHost	null	Camel 2.11:Producer only The http proxy Host url which will be used by <code>Jetty</code> client.
proxyPort	null	Camel 2.11:Producer only The http proxy port which will be used by <code>Jetty</code> client.
sendServerVersion	true	Camel 2.13: if the option is true, <code>jetty</code> will send the server header with the <code>jetty</code> version information to the client which sends the request. NOTE please make sure there is no any other camel-jetty endpoint is share the same port, otherwise this option may not work as expected.
SendDateHeader	false	Camel 2.14: if the option is true, <code>jetty</code> server will send the date header to the client which sends the request. NOTE please make sure there is no any other camel-jetty endpoint is share the same port, otherwise this option may not work as expected.

3.37.3. Message Headers

Camel uses the same message headers as the [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.

3.37.4. 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](#)

3.37.5. Component Options

The `JettyHttpComponent` provides the following options:

Name	Default Value	Description
<code>enableJmx</code>	<code>false</code>	If this option is true, Jetty JMX support will be enabled for this endpoint. See Jetty JMX support for more details.
<code>sslKeyPassword</code>	<code>null</code>	Consumer only : The password for the keystore when using SSL.
<code>sslPassword</code>	<code>null</code>	Consumer only : The password when using SSL.
<code>sslKeystore</code>	<code>null</code>	Consumer only : The path to the keystore.
<code>minThreads</code>	<code>null</code>	Consumer only : To set a value for minimum number of threads in server thread pool. Note that both a min and max size must be configured.
<code>maxThreads</code>	<code>null</code>	Consumer only : To set a value for maximum number of threads in server thread pool. Note that both a min and max size must be configured.
<code>threadPool</code>	<code>null</code>	Consumer only : To use a custom thread pool for the server. This option is only needed in special circumstances.
<code>sslSocketConnectors</code>	<code>null</code>	Consumer only : A map which contains per port number specific SSL connectors. See section <i>SSL support</i> for more details.
<code>socketConnectors</code>	<code>null</code>	Consumer only : A map which contains per port number specific HTTP connectors. Uses the same principle as <code>sslSocketConnectors</code> and therefore see section <i>SSL support</i> for more details.
<code>sslSocketConnector-Properties</code>	<code>null</code>	Consumer only . A map which contains general SSL connector properties. See section <i>SSL support</i> for more details.
<code>socketConnector-Properties</code>	<code>null</code>	Consumer only . A map which contains general HTTP connector properties. Uses the same principle as <code>sslSocketConnectorProperties</code> and therefore see section <i>SSL support</i> for more details.
<code>httpClient</code>	<code>null</code>	(Deprecated) Producer only : To use a custom <code>HttpClient</code> with the jetty producer. This option is removed from Camel 2.11 onwards, instead you can set the option on the endpoint instead.
<code>httpClientMinThreads</code>	<code>null</code>	Producer only : To set a value for minimum number of threads in <code>HttpClient</code> thread pool. Note that both a min and max size must be configured.
<code>httpClientMaxThreads</code>	<code>null</code>	Producer only : To set a value for maximum number of threads in <code>HttpClient</code> thread pool. Note that both a min and max size must be configured.
<code>httpClientThreadPool</code>	<code>null</code>	(Deprecated) Producer only : To use a custom thread pool for the client. This option will be removed starting with Camel 2.11.
<code>sslContextParameters</code>	<code>null</code>	To configure a custom SSL/TLS configuration options at the component level.
<code>requestBufferSize</code>	<code>null</code>	Camel 2.11.2 : Allows to configure a custom value of the request buffer size on the Jetty connectors.

Name	Default Value	Description
requestHeaderSize	null	Camel 2.11.2: Allows to configure a custom value of the request header size on the Jetty connectors.
responseBufferSize	null	Camel 2.11.2: Allows to configure a custom value of the response buffer size on the Jetty connectors.
responseHeaderSize	null	Camel 2.11.2: Allows to configure a custom value of the response header size on the Jetty connectors.
proxyHost	null	Camel 2.12.2/2.11.3 To use a http proxy.
proxyPort	null	Camel 2.12.2/2.11.3: To use a http proxy.

3.37.6. 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());
```



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.

3.37.7. 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();
    ...
}
```

3.37.8. 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.

3.37.8.1. 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 explicitly configured as above with the port number as entry).

```
<bean id="jetty"
  class="org.apache.camel.component.jetty.JettyHttpComponent">
  <property name="sslSocketConnectorProperties">
    <map>
      <entry name="password" value="..." />
      <entry name="keyPassword" value="..." />
      <entry name="keystore" value="..." />
      <entry name="needClientAuth" value="..." />
      <entry name="truststore" value="..." />
    </map>
  </property>
</bean>
```

3.37.8.2. 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">
```

```

    <map>
      <entry key="acceptors" value="4" />
      <entry key="maxIdleTime" value="300000" />
    </map>
  </property>
</bean>

```

3.37.8.3. 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 `Exchange.HTTP_RESPONSE_CODE` header of the OUT message.

3.37.8.4. 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.

3.38. JGroups

JGroups is a toolkit for reliable multicast communication. The **jgroups** component provides exchange of messages between Camel infrastructure and **JGroups** clusters.

Maven users will need to add the following dependency to their `pom.xml` for this component.

```

<dependency>
  <groupId>org.apache-extras.camel-extra</groupId>
  <artifactId>camel-jgroups</artifactId>
  <!-- use the same version as your Camel core version -->
  <version>x.y.z</version>
</dependency>

```

```
</dependency>
```

Starting from the Camel **2.13.0**, JGroups component has been moved from Camel Extra under the umbrella of the Apache Camel. If you are using Camel **2.13.0** or higher, please use the following POM entry instead.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jgroups</artifactId>
  <!-- use the same version as your Camel core version -->
  <version>x.y.z</version>
</dependency>
```

3.38.1. URI format

```
jgroups:clusterName[?options]
```

Where **clusterName** represents the name of the JGroups cluster the component should connect to.

3.38.2. Options

Name	Default Value	Description
channelProperties	null	Camel 2.10.0: Specifies configuration properties of the JChannel used by the endpoint.
enableViewMessages	false	Camel 2.13.0: Consumer only. If set to true, the consumer endpoint will receive <code>org.jgroups.View</code> messages as well (not only <code>org.jgroups.Message</code> instances). By default only regular messages are consumed by the endpoint.

3.38.3. Headers

Header	Constant	Since version	Description
JGROUPS_ORIGINAL_MESSAGE	JGroupsEndpoint.HEADER_ JGROUPS_ORIGINAL_MESSAGE	2.13.0	The original <code>org.jgroups.Message</code> instance from which the body of the consumed message has been extracted.
JGROUPS_SRC	JGroupsEndpoint.HEADER_ JGROUPS_SRC	2.10.0	Consumer: The <code>org.jgroups.Address</code> instance extracted by <code>org.jgroups.Message.getSrc()</code> method of the consumed message. Producer: The custom source <code>org.jgroups.Address</code> of the message to be sent.
JGROUPS_DEST	JGroupsEndpoint.HEADER_ JGROUPS_DEST	2.10.0	Consumer: The <code>org.jgroups.Address</code> instance extracted by <code>org.jgroups.Message.getDest()</code> method of the consumed message. Producer: The custom destination <code>org.jgroups.Address</code> of the message to be sent.

Header	Constant	Since version	Description
JGROUPS_CHANNEL _ADDRESS	JGroupsEndpoint.HEADER_ JGROUPS_CHANNEL_ADDRESS	2.13.0	Address (<code>org.jgroups.Address</code>) of the channel associated with the endpoint.

3.38.4. Usage

Using `jgroups` component on the consumer side of the route will capture messages received by the `JChannel` associated with the endpoint and forward them to the Camel route. `JGroups` consumer processes incoming messages [asynchronously](#).

```
// Capture messages from cluster named
// 'clusterName' and send them to Camel route.
from("jgroups:clusterName").to("seda:queue");
```

Using `jgroups` component on the producer side of the route will forward body of the Camel exchanges to the `JChannel` instance managed by the endpoint.

```
// Send message to the cluster named 'clusterName'
from("direct:start").to("jgroups:clusterName");
```

3.38.5. Predefined filters

Starting from version **2.13.0** of Camel, `JGroups` component comes with predefined filters factory class named `JGroupsFilters`.

If you would like to consume only view changes notifications sent to coordinator of the cluster (and ignore these sent to the "slave" nodes), use the `JGroupsFilters.dropNonCoordinatorViews()` filter. This filter is particularly useful when you want a single Camel node to become the master in the cluster, because messages passing this filter notifies you when given node has become a coordinator of the cluster. The snippet below demonstrates how to collect only messages received by the master node.

```
import static org.apache.camel.component.jgroups.JGroupsFilters.dropNonCoordinatorViews;
...
from("jgroups:clusterName?enableViewMessages=true").
    filter(dropNonCoordinatorViews()).
    to("seda:masterNodeEventsQueue");
```

3.38.6. Predefined expressions

Starting from version **2.13.0** of Camel, `JGroups` component comes with predefined expressions factory class named `JGroupsExpressions`.

If you would like to create [delayer](#) that would affect the route only if the Camel context has not been started yet, use the `JGroupsExpressions.delayIfContextNotStarted(long delay)` factory method. The expression created by this factory method will return given delay value only if the Camel context is in the state different than `started`. This expression is particularly useful if you would like to use `JGroups` component for keeping singleton (master) route within the cluster. [Control Bus](#) `start` command won't initialize the singleton route if the Camel Context hasn't been yet started. So you need to delay a startup of the master route, to be sure that it has been initialized after the Camel Context startup. Because such scenario can happen only during the initialization

of the cluster, we don't want to delay startup of the slave node becoming the new master - that's why we need a conditional delay expression.

The snippet below demonstrates how to use conditional delaying with the JGroups component to delay the initial startup of master node in the cluster.

```
import static java.util.concurrent.TimeUnit.SECONDS;
import static
    org.apache.camel.component.jgroups.JGroupsExpressions.delayIfContextNotStarted;
import static org.apache.camel.component.jgroups.JGroupsFilters.dropNonCoordinatorViews;
...
from("jgroups:clusterName?enableViewMessages=true").
    filter(dropNonCoordinatorViews()).
    threads().delay(delayIfContextNotStarted(SECONDS.toMillis(5))). // run in separated
    and delayed thread. Delay only if the context hasn't been started already.
    to("controlbus:route?routeId=masterRoute&action=start&async=true");

from("timer://master?
repeatCount=1").routeId("masterRoute").autoStartup(false).to(masterMockUri);
```

3.39. JMS



If you are using [Apache ActiveMQ](#), you should prefer the [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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.39.1. 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&...`

3.39.2. 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 [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` = false
- `CACHE_NONE` = if `transacted` = 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 try to name your headers to be 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.



Are you using transactions? If you are consuming messages, and have `transacted=true`, then the default settings for cache level can impact performance. The default setting is always `CACHE_CONSUMER`. However, with the `CACHE_AUTO` setting, when you use transactions the cache level is effectively set to `CACHE_NONE`, appropriate for transactions.

3.39.3. 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.

3.39.3.1. 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. Starting with Camel 2.11, this option can also be used when doing request/reply over JMS . See also the <code>maxMessagesPerTask</code> option to control dynamic scaling up/down of threads.
disableReplyTo	false	If true, a producer will behave like a <code>InOnly</code> exchange with the exception that <code>JMSReplyTo</code> header is sent out and not be suppressed like in the case of <code>InOnly</code> . Like <code>InOnly</code> the producer will not wait for a reply. A consumer with this flag will behave like <code>InOnly</code> . This feature can be used to bridge <code>InOut</code> requests to another queue so that a route on the other queue will send its response directly back to the original <code>JMSReplyTo</code> .
durableSubscriptionName	null	The durable subscriber name for specifying durable topic subscriptions. The <code>clientId</code> option must be configured as well.
maxConcurrentConsumers	1	Specifies the maximum number of concurrent consumers. Starting with Camel 2.11, this option can also be used when doing request/reply over JMS . See also the <code>maxMessagesPerTask</code> option to control dynamic scaling up/down of threads.
maxMessagesPerTask	-1	The number of messages per task, -1 for unlimited. If you use a range for concurrent consumers (e.g. <code>min < max</code>), then this option can be used to set a value to e.g. 100 to control how fast the consumers will shrink when less work is required.
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 <code>JMSPriority</code> , <code>JMSDeliveryMode</code> , and <code>JMSExpiration</code> . 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 <code>explicitQosEnabled</code> option, by contrast, will only use options set on the endpoint, and not values from the message header.
replyTo	null	Provides an explicit <code>ReplyTo</code> destination, which overrides any incoming value of <code>Message.getJMSReplyTo()</code> . If you do [Request Reply] over JMS then read the Camel Request-reply over JMS section for more details.
replyToType	null	Allows to explicit specify which kind of strategy to use for <code>replyTo</code> queues when doing request/reply over JMS . Possible values are: <code>{{Temporary}}</code> , <code>{{Shared}}</code> , or <code>{{Exclusive}}</code> . By default Camel will use temporary queues. However if <code>{{replyTo}}</code> has been configured, then <code>{{Shared}}</code> 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). The default is 20 seconds. From Camel 2.13/2.12.3 onwards you can include the header <code>"CamelJmsRequestTimeout"</code> to override this endpoint configured timeout value, and thus have per message

Option	Default Value	Description
		individual timeout values. See below in section <i>About time to live</i> for more details. See also the <i>requestTimeoutCheckerInterval</i> 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 this option, if you start and stop JMS routes at runtime, while there are still messages enqueued on the queue. If this option is <code>false</code> , 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: <code>SESSION_TRANSACTED</code> , <code>CLIENT_ACKNOWLEDGE</code> , <code>AUTO_ACKNOWLEDGE</code> , <code>DUPS_OK_ACKNOWLEDGE</code>
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 <code>acknowledgementModeName</code> instead.
allowNullBody	true	Whether to allow sending messages with no body. If this option is <code>false</code> and the message body is null, then an <code>JMSEException</code> is thrown.
alwaysCopyMessage	false	If <code>true</code> , 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 <code>replyToDestinationSelectorName</code> is set (incidentally, Camel will set the <code>alwaysCopyMessage</code> option to <code>true</code> , if a <code>replyToDestinationSelectorName</code> is set)
asyncStartListener	false	Whether to startup the <code>JmsConsumer</code> message listener asynchronously, when starting a route. For example if a <code>JmsConsumer</code> 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 <code>true</code> , you will let routes startup, while the <code>JmsConsumer</code> 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 <code>WARN</code> level, and the consumer will not be able to receive messages; You can then restart the route to retry.
asyncStopListener	false	Whether to stop the <code>JmsConsumer</code> message listener asynchronously, when stopping a route.

Option	Default Value	Description
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 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	null	Sets the cache level by ID for the underlying JMS resources. See cacheLevelName for more details.
consumerType	Default	<p>The consumer type to use, which can be one of: Simple, Default or Custom. The consumer type determines which Spring JMS listener to use.</p> <ul style="list-style-type: none"> • Default will use <code>org.springframework.jms.listener.DefaultMessageListenerContainer</code> • Simple will use <code>org.springframework.jms.listener.SimpleMessageListenerContainer</code> • When Custom is specified, the <code>MessageListenerContainerFactory</code> defined by the <code>messageListener-ContainerFactoryRef</code> option will determine what <code>AbstractMessage-ListenerContainer</code> to use.
connectionFactory	null	The default JMS connection factory to use for the <code>listenerConnectionFactory</code> and <code>templateConnectionFactory</code> , if neither is specified.
defaultTaskExecutorType		Specifies what default TaskExecutor type to use in the <code>DefaultMessageListenerContainer</code> , for both consumer endpoints and the ReplyTo consumer of producer endpoints. Possible values: <code>SimpleAsync</code> (uses Spring's SimpleAsyncTaskExecutor) or <code>ThreadPool</code> (uses Spring's ThreadPoolTaskExecutor with optimal values - cached threadpool-like). If not set, it defaults to a cached thread pool for consumer endpoints and <code>SimpleAsync</code> for reply consumers. The use of <code>ThreadPool</code> is recommended to reduce "thread trash" in elastic configurations with dynamically increasing and decreasing concurrent consumers.
deliveryMode	null	Camel 2.12.2/2.13: Specifies the delivery mode to be used. Possibles values are those defined by <code>javax.jms.DeliveryMode</code> .
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.

Option	Default Value	Description
<code>destinationResolver</code>	<code>null</code>	A pluggable <code>org.springframework.jms.support.destination.DestinationResolver</code> that allows you to use your own resolver (for example, to lookup the real destination in a JNDI registry).
<code>disableTimeToLive</code>	<code>false</code>	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 <code>{{requestTimeout}}</code> value as time to live on the message being sent. The problem is that the sender and receiver systems have to have their clocks synchronized, so they are in sync. This is not always so easy to archive. So you can use <code>{{disableTimeToLive=true}}</code> to <code>*not*</code> set a time to live value on the send message. Then the message will not expire on the receiver system.
<code>eagerLoadingOfProperties</code>	<code>false</code>	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.
<code>exceptionListener</code>	<code>null</code>	Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.
<code>errorHandler</code>	<code>null</code>	Specifies a <code>org.springframework.util.ErrorHandler</code> 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. 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.
<code>errorHandlerLoggingLevel</code>	<code>WARN</code>	Allows for configuring the default errorHandler logging level for logging uncaught exceptions.
<code>errorHandlerLogStack-Trace</code>	<code>true</code>	Allows to control whether stacktraces should be logged or not, by the default errorHandler.
<code>explicitQosEnabled</code>	<code>false</code>	Set if the <code>deliveryMode</code> , <code>priority</code> or <code>timeToLive</code> qualities of service should be used when sending messages. This option is based on Spring's <code>JmsTemplate</code> . The <code>deliveryMode</code> , <code>priority</code> and <code>timeToLive</code> options are applied to the current endpoint. This contrasts with the <code>preserveMessageQos</code> option, which operates at message granularity, reading QoS properties exclusively from the Camel In message headers.
<code>exposeListenerSession</code>	<code>true</code>	Specifies whether the listener session should be exposed when consuming messages.
<code>forceSendOriginalMessage</code>	<code>false</code>	When using <code>mapJmsMessage=false</code> 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 <code>true</code> to force Camel to send the original JMS message that was received.
<code>idleConsumerLimit</code>	<code>1</code>	Specify the limit for the number of consumers that are allowed to be idle at any given time.
<code>idleTaskExecutionLimit</code>	<code>1</code>	Specifies the limit for idle executions of a receive task, not having received any message within its 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 <code>maxConcurrentConsumers</code> setting). There is additional doc available from Spring .
<code>idleConsumerLimit</code>	<code>1</code>	Specify the limit for the number of consumers that are allowed to be idle at any given time.

Option	Default Value	Description
includeSentJMSMessageID	false	Only applicable when sending to JMS destination using InOnly (eg fire and forget). Enabling this option will enrich the Camel Exchange with the actual JMSMessageID that was used by the JMS client when the message was sent to the JMS destination.
includeAllJMSXProperties	false	Camel 2.11.2/2.12: Whether to include all JMSXxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSXAppID, and JMSXUserID etc. Note: If you are using a custom headerFilterStrategy then this option does not apply.
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 appropriate payload type, such as javax.jms.TextMessage to a String etc. See section about how mapping works below for more details.
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.
messageIdEnabled	true	When sending, specifies whether message IDs should be added.
messageListener-ContainerFactoryRef	null	Registry ID of the MessageListenerContainerFactory used to determine what AbstractMessageListenerContainer to use to consume messages. Setting this will automatically set consumerType to Custom.
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).

Option	Default Value	Description
		The <code>explicitQosEnabled</code> option must also be enabled in order for this option to have any effect.
<code>pubSubNoLocal</code>	<code>false</code>	Specifies whether to inhibit the delivery of messages published by its own connection.
<code>receiveTimeout</code>	<i>None</i>	The timeout for receiving messages (in milliseconds).
<code>recoveryInterval</code>	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.
<code>replyToCacheLevelName</code>	CACHE_CONSUMER	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: <code>CACHE_CONSUMER</code> for exclusive or shared w/ <code>{{replyToSelectorName}}</code> . And <code>CACHE_SESSION</code> for shared without <code>replyToSelectorName</code> . Some JMS brokers such as IBM WebSphere may require to set the <code>replyToCacheLevelName=CACHE_NONE</code> to work. Note: If using temporary queues then <code>CACHE_NONE</code> is not allowed, and you must use a higher value such as <code>CACHE_CONSUMER</code> or <code>CACHE_SESSION</code> .
<code>replyToDestination-SelectorName</code>	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).
<code>replyToDelivery-Persistent</code>	<code>true</code>	Specifies whether to use persistent delivery by default for replies.
<code>requestTimeout-CheckerInterval</code>	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 interval, to check more frequently. The timeout is determined by the <code>requestTimeout</code> option.
<code>subscriptionDurable</code>	<code>false</code>	@deprecated: Enabled by default, if you specify a <code>durableSubscriptionName</code> and a <code>clientId</code> .
<code>taskExecutor</code>	null	Allows you to specify a custom task executor for consuming messages.
<code>taskExecutorSpring2</code>	null	To use when using Spring 2.x with Camel. Allows you to specify a custom task executor for consuming messages.
<code>templateConnection-Factory</code>	null	The JMS connection factory used for sending messages.
<code>transactedInOut</code>	<code>false</code>	@deprecated: Specifies whether to use transacted mode for sending messages using the InOut Exchange Pattern . Applies only to producer endpoints. See Enabling Transacted Consumption on the Camel website for more details.
<code>transactionManager</code>	null	The Spring transaction manager to use.
<code>transactionName</code>	JmsConsumer [destination-Name]	The name of the transaction to use.
<code>transactionTimeout</code>	null	The timeout value of the transaction, if using transacted mode.
<code>transferException</code>	<code>false</code>	If enabled and you are using Request Reply messaging (InOut) and an Exchange failed on the consumer side, then the caused <code>Exception</code> will be send back in response as a <code>javax.jms.ObjectMessage</code> . If the client is Camel, the returned <code>Exception</code> is rethrown. This allows you to use Camel 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 <code>Exception</code> on the consumer side can be wrapped in an outer exception such as

Option	Default Value	Description
		<code>org.apache.camel.RuntimeCamelException</code> when returned to the producer.
<code>transferExchange</code>	<code>false</code>	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 <code>WARN</code> level. You <i>must</i> enable this option on both the producer and consumer side, so Camel knows the payloads form an Exchange and not a regular payload.
<code>username</code>	<code>null</code>	The username for the connector factory.
<code>useMessageIDs-as-CorrelationID</code>	<code>false</code>	Specifies whether <code>JMSMessageID</code> should always be used as <code>JMSCorrelationID</code> for InOut messages.

Message Mapping between JMS and Camel Camel 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
<code>String</code>	<code>javax.jms.TextMessage</code>	
<code>org.w3c.dom.Node</code>	<code>javax.jms.TextMessage</code>	The DOM will be converted to <code>String</code> .
<code>Map</code>	<code>javax.jms.MapMessage</code>	
<code>java.io.Serializable</code>	<code>javax.jms.ObjectMessage</code>	
<code>byte[]</code>	<code>javax.jms.BytesMessage</code>	
<code>java.io.File</code>	<code>javax.jms.BytesMessage</code>	
<code>java.io.Reader</code>	<code>javax.jms.BytesMessage</code>	
<code>java.io.InputStream</code>	<code>javax.jms.BytesMessage</code>	
<code>java.nio.ByteBuffer</code>	<code>javax.jms.BytesMessage</code>	

When receiving a JMS message, Camel converts the JMS message to the following body type:

JMS Message	Body Type
<code>javax.jms.TextMessage</code>	<code>String</code>
<code>javax.jms.BytesMessage</code>	<code>byte[]</code>
<code>javax.jms.MapMessage</code>	<code>Map<String, Object></code>
<code>javax.jms.ObjectMessage</code>	<code>Object</code>

3.39.4. 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 and the reverse when consuming JMS messages:
 - . is replaced by `_DOT_` and the reverse replacement when Camel consumes the message.
 - is replaced by `_HYPHEN_` and the reverse replacement when Camel consumes the message.

- 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}
```

3.39.5. Message format when receiving

Camel adds the following properties to the `Exchange` when it receives a message:

Property	Type	Description
<code>org.apache.camel.jms.replyDestination</code>	<code>javax.jms.Destination</code>	The reply destination.

Camel adds the following JMS properties to the In message headers when it receives a JMS message:

Header	Type	Description
<code>JMSCorrelationID</code>	<code>String</code>	The JMS correlation ID.
<code>JMSDeliveryMode</code>	<code>int</code>	The JMS delivery mode.
<code>JMSDestination</code>	<code>javax.jms.Destination</code>	The JMS destination.
<code>JMSExpiration</code>	<code>long</code>	The JMS expiration.
<code>JMSMessageID</code>	<code>String</code>	The JMS unique message ID.
<code>JMSPriority</code>	<code>int</code>	The JMS priority (with 0 as the lowest priority and 9 as the highest).
<code>JMSRedelivered</code>	<code>boolean</code>	the JMS message redelivered.
<code>JMSReplyTo</code>	<code>javax.jms.Destination</code>	The JMS reply-to destination.
<code>JMSTimestamp</code>	<code>long</code>	The JMS timestamp.
<code>JMSType</code>	<code>String</code>	The JMS type.
<code>JMSXGroupID</code>	<code>String</code>	The JMS group ID.

As all the above information is standard JMS you can check the [JMS documentation](#) for further details.

3.39.6. About using Camel to send and receive messages and `JMSReplyTo`

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.

3.39.6.1. JmsProducer

The `JmsProducer` behaves as follows, depending on configuration:

Exchange Pattern	Other options	Description
<i>InOut</i>	-	Camel will expect a reply, set a temporary <code>JMSReplyTo</code> , and after sending the message, it will start to listen for the reply message on the temporary queue.
<i>InOut</i>	<code>JMSReplyTo</code> is set	Camel will expect a reply and, after sending the message, it will start to listen for the reply message on the specified <code>JMSReplyTo</code> queue.
<i>InOnly</i>	-	Camel will send the message and not expect a reply.
<i>InOnly</i>	<code>JMSReplyTo</code> is set	By default, Camel discards the <code>JMSReplyTo</code> destination and clears the <code>JMSReplyTo</code> header before sending the message. Camel then sends the message and does not expect a reply. Camel logs this in the log at <code>DEBUG</code> level. You can use <code>preserveMessageQos=true</code> to instruct Camel to keep the <code>JMSReplyTo</code> . In all situations the <code>JmsProducer</code> does not expect any reply and thus continue after sending the message.

3.39.6.2. JmsConsumer

The `JmsConsumer` behaves as follows, depending on configuration:

Exchange Pattern	Other options	Description
<i>InOut</i>	-	Camel will send the reply back to the <code>JMSReplyTo</code> queue.
<i>InOnly</i>	-	Camel will not send a reply back, as the pattern is <i>InOnly</i> .
-	<code>disableReplyTo=true</code>	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 [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" );
```

3.39.7. 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.

3.39.8. Samples

JMS is used in many examples for other components as well. But we provide a few samples below to get started.

3.39.8.1. 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");

```

3.39.8.2. 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");

```

3.39.8.3. Using Annotations

Camel also has annotations so you can use [POJO Consuming](#) and [POJO Producing](#).

3.39.8.4. 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>
```

3.39.8.5. 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](#).

3.40. JMX

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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.40.1. URI Format and Options

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&...`, where *option* can be:

Table 3.12.

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 <code>NotificationFilter</code> . The <code>#ref</code> 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"
testConnection-OnStartup		true	Starting with Camel 2.11, if true, the consumer will throw an exception when unable to establish the JMX connection upon startup. If false, the consumer will attempt to establish the JMX connection every 'x' seconds until the connection is made - where 'x' is the configured using the <code>reconnectDelay</code> option.
reconnectOn-ConnectionFailure		false	Starting with Camel 2.11, if true, the consumer will attempt to reconnect to the JMX server when any connection failure occurs. The consumer will attempt to re-establish the JMX connection every 'x' seconds until the connection is made--where 'x' is the configured using the <code>reconnectDelay</code> option.
reconnectDelay		10 seconds	Starting with Camel 2.11, the number of seconds to wait before retrying creation of the initial connection or before reconnecting a lost connection.

3.40.2. 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."

3.40.3. 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.

3.40.4. 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.

3.40.5. Example

```
from("jmx:platform?objectDomain=jmxExample&key.name=simpleBean").
to("log:jmxEvent");
```

A full example is [here](#).

3.41. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.41.1. 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 [Message Translator](#) in front of the endpoint to perform the necessary conversion first.

3.41.2. 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 (and when routing is done), 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 it has been processed (and when routing is done).

From **Camel 2.13** onwards you can use [@PreConsumed](#) which will be invoked on your entity bean before it has been processed (before routing).

3.41.3. 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&...`

3.41.4. Options

Name	Default Value	Description
<code>entityType</code>	<code>entityClassName</code>	Overrides the <code>entityClassName</code> from the URI.
<code>persistenceUnit</code>	<code>camel</code>	The JPA persistence unit used by default.
<code>consumeDelete</code>	<code>true</code>	JPA consumer only: If <code>true</code> , the entity is deleted after it is consumed; if <code>false</code> , the entity is not deleted.
<code>consumeLockEntity</code>	<code>true</code>	JPA consumer only: Specifies whether or not to set an exclusive lock on each entity bean while processing the results from polling.
<code>flushOnSend</code>	<code>true</code>	JPA producer only: Flushes the EntityManager after the entity bean has been persisted.
<code>maximumResults</code>	<code>-1</code>	JPA consumer only: Set the maximum number of results to retrieve on the Query .
<code>transactionManager</code>	<code>null</code>	This option is Registry based which requires the <code>#</code> notation so that the given <code>transactionManager</code> being specified can be looked up properly, e.g. <code>transactionManager=#myTransactionManager</code> . It specifies the transaction manager to use. If none provided, Camel will use a <code>JpaTransactionManager</code> by default. Can be used to set a JTA transaction manager (for integration with an EJB container).
<code>consumer.delay</code>	<code>500</code>	JPA consumer only: Delay in milliseconds between each poll.
<code>consumer.initialDelay</code>	<code>1000</code>	JPA consumer only: Milliseconds before polling starts.
<code>consumer.useFixedDelay</code>	<code>false</code>	JPA consumer only: Set to <code>true</code> to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
<code>maxMessagesPerPoll</code>	<code>0</code>	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.
<code>consumer.query</code>		JPA consumer only: To use a custom query when consuming data.
<code>consumer.namedQuery</code>		JPA consumer only: To use a named query when consuming data.
<code>consumer.nativeQuery</code>		JPA consumer only: To use a custom native query when consuming data. You may want to use the option <code>consumer.resultClass</code> also when using native queries.
<code>consumer.parameters</code>		Camel 2.12: JPA consumer only: This option is Registry based which requires the <code>#</code> notation. This key/value mapping is used for building the query parameters. It's expected to be of the generic type <code>java.util.Map<String, Object></code> where the keys are the named parameters of a given JPA query and the values are their corresponding effective values you want to select for.
<code>consumer.resultClass</code>		Camel 2.7: JPA consumer only: Defines the type of the returned payload (we will call <code>entityManager.createNativeQuery(nativeQuery, resultClass)</code> instead of <code>entityManager.createNativeQuery(nativeQuery)</code>). Without this option, we will return an

Name	Default Value	Description
		object array. Only has an affect when using in conjunction with native query when consuming data.
<code>consumer.transacted</code>	false	<i>Camel 2.7.5/2.8.3/2.9:</i> JPA consumer only: Whether to run the consumer in transacted mode, by which all messages will either commit or rollback, when the entire batch has been processed. The default behavior (false) is to commit all the previously successfully processed messages, and only rollback the last failed message.
<code>consumer.lockModeType</code>	WRITE	<i>Camel 2.11.2/2.12:</i> To configure the lock mode on the consumer. The possible values is defined in the enum <code>javax.persistence.LockModeType</code> . The default value is changed to <code>PESSIMISTIC_WRITE</code> since <i>Camel 2.13</i> .
<code>consumer.SkipLockedEntity</code>	false	<i>Camel 2.13:</i> To configure whether to use NOWAIT on lock and silently skip the entity.
<code>usePersist</code>	false	JPA producer only: Indicates to use <code>entityManager.persist(entity)</code> instead of <code>entityManager.merge(entity)</code> . Note: <code>entityManager.persist(entity)</code> doesn't work for detached entities (where the EntityManager has to execute an UPDATE instead of an INSERT query)!
<code>joinTransaction</code>	true	<i>Camel 2.12.3:</i> camel-jpa will join transaction by default from Camel 2.12 onwards. You can use this option to turn this off, for example if you use <code>LOCAL_RESOURCE</code> and join transaction doesn't work with your JPA provider. This option can also be set globally on the <code>JpaComponent</code> , instead of having to set it on all endpoints.
<code>usePassedInEntityManager</code>	false	<i>Camel 2.12.4/2.13.1 JPA producer only:</i> If set to true, then Camel will use the EntityManager from the header <code>JpaConstants.ENTITYMANAGER</code> instead of the configured entity manager on the component/endpoint. This allows end users to control which entity manager will be in use.

3.41.5. Message Headers

Camel adds the following message headers to the exchange:

Header	Type	Description
<code>CamelJpaTemplate</code>	<code>JpaTemplate</code>	Not supported anymore since Camel 2.12: The <code>JpaTemplate</code> 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. See CAMEL-5932 for the reason why the support for this header has been dropped.
<code>CamelEntityManager</code>	<code>EntityManager</code>	Camel 2.12: JPA consumer / Camel 2.12.2: JPA producer: The JPA <code>EntityManager</code> object being used by <code>JpaConsumer</code> or <code>JpaProducer</code> .

3.41.6. 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**.

3.41.7. Configuring TransactionManager

Since **Camel 2.3** the `JpaComponent` will auto lookup the `TransactionManager` from the [Registry](#). If Camel won't find any `TransactionManager` instance registered, it will also look up for the `TransactionTemplate` and try to extract `TransactionManager` from it.

If none `TransactionTemplate` is available in the registry, `JpaEndpoint` will auto create their own instance of `TransactionManager` which most often is not what you want.

If more than single instance of the `TransactionManager` is found, Camel will log a **WARN**. In such cases you might want to instantiate and explicitly configure 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>
```

3.41.8. 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");
```

3.41.9. 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");
```

3.41.10. 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.

3.41.11. Example

See [Tracer Example](#) for an example using [JPA](#) to store traced messages into a database.

3.42. Kafka

Available as of Camel 2.13

The **kafka** component is used for communicating with [Apache Kafka](#) message broker.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-kafka</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.42.1. URI format

```
kafka:server:port[?options]
```

3.42.2. Options

Property	Default	Description
<code>zookeeperHost</code>		The zookeeper host to use
<code>zookeeperPort</code>	2181	The zookeeper port to use

Property	Default	Description
zookeeperConnect		Camel 2.13.3/2.14.1: If in use, then zookeeperHost/zookeeperPort is not used.
topic		The topic to use
groupId		
partitioner		
consumerStreams	10	
clientId		
zookeeperSessionTimeoutMs		
zookeeperConnectionTimeoutMs		
zookeeperSyncTimeMs		

You can append query options to the URI in the following format, ?option=value&option=value&...

3.42.3. Producer Options

Property	Default	Description
producerType		
compressionCodec		
compressedTopics		
messageSendMaxRetries		
retryBackoffMs		
topicMetadataRefreshIntervalMs		
sendBufferBytes		
requestRequiredAcks		
requestTimeoutMs		
queueBufferingMaxMs		
queueBufferingMaxMessages		
queueEnqueueTimeoutMs		
batchNumMessages		
serializerClass		
keySerializerClass		

3.42.4. Consumer Options

Property	Default	Description
consumerId		
socketTimeoutMs		
socketReceiveBufferBytes		
fetchMessageMaxBytes		
autoCommitEnable		
autoCommitIntervalMs		
queuedMaxMessages		
rebalanceMaxRetries		

Property	Default	Description
fetchMinBytes		
fetchWaitMaxMs		
rebalanceBackoffMs		
refreshLeaderBackoffMs		
autoOffsetReset		
consumerTimeoutMs		

3.42.5. Samples

Consuming messages:

```
from( "kafka:localhost:9092?
topic=test&zookeeperHost=localhost&zookeeperPort=2181&groupId=group1" )
.to( "log:input" );
```

3.42.6. Endpoints

Camel supports the [Message Endpoint](#) pattern using the [Endpoint](#) interface. Endpoints are usually created by a [Component](#) and Endpoints are usually referred to in the [DSL](#) via their [URIs](#).

From an Endpoint you can use the following methods

- [createProducer\(\)](#) will create a [Producer](#) for sending message exchanges to the endpoint
- [createConsumer\(\)](#) implements the [Event Driven Consumer](#) pattern for consuming message exchanges from the endpoint via a [Processor](#) when creating a [Consumer](#)
- [createPollingConsumer\(\)](#) implements the [Polling Consumer](#) pattern for consuming message exchanges from the endpoint via a [PollingConsumer](#)

3.43. Krati

Available as of Camel 2.9

This component allows the use krati datastores and datasets inside Camel. Krati is a simple persistent data store with very low latency and high throughput. It is designed for easy integration with read-write-intensive applications with little effort in tuning configuration, performance and JVM garbage collection.

Camel provides a producer and consumer for krati datastore_(key/value engine)_. It also provides an idempotent repository for filtering out duplicate messages.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-krati</artifactId>
  <version>x.x.x</version>
```

```
<!-- use the same version as your Camel core version -->
</dependency>
```

3.43.1. URI format

```
krati:[the path of the datastore][?options]
```

The **path of the datastore** is the relative path of the folder that krati will use for its datastore.

You can append query options to the URI in the following format, ?option=value&option=value&...

3.43.2. Krati URI Options

Name	Default Value	Description
operation	CamelKratiPut	Producer Only. Specifies the type of operation that will be performed to the datastore. Allowed values are CamelKratiPut, CamelKratiGet, CamelKratiDelete & CamelKratiDeleteAll.
initialCapacity	100	The initial capacity of the store.
keySerializer	KratiDefaultSerializer	The serializer that will be used to serialize the key.
valueSerializer	KratiDefaultSerializer	The serializer that will be used to serialize the value.
segmentFactory	ChannelSegmentFactory	The segment factory to use. Allowed instance classes: ChannelSegmentFactory, MemorySegmentFactory, MappedSegmentFactory & WriteBufferSegmentFactory.
hashFunction	FnvHashFunction	The hash function to use. Allowed instance classes: FnvHashFunction, Fnv1Hash32, FnvHash64, Fnv1aHash32, Fnv1aHash64, JenkinsHashFunction, MurmurHashFunction
maxMessagesPerPoll		Camel 2.10.5/2.11.1: The maximum number of messages which can be received in one poll. This can be used to avoid reading in too much data and taking up too much memory.

You can have as many of these options as you like.

```
krati:/tmp/krati?operation=CamelKratiGet&initialCapacity=10000&keySerializer=
#myCustomSerializer
```

For producer endpoint you can override all of the above URI options by passing the appropriate headers to the message.

3.43.2.1. Message Headers for datastore

Header	Description
CamelKratiOperation	The operation to be performed on the datastore. The valid options are <ul style="list-style-type: none"> CamelKratiAdd

Header	Description
	<ul style="list-style-type: none"> • CamelKratiGet • CamelKratiDelete • CamelKratiDeleteAll
CamelKratiKey	The key
CamelKratiValue	The value

3.43.3. Usage Samples

3.43.3.1. Example 1: Putting to the datastore.

This example will show you how you can store any message inside a datastore.

```
from("direct:put").to("krati:target/test/producertest");
```

In the above example you can override any of the URI parameters with headers on the message.

```
<route>
  <from uri="direct:put"/>
  <to uri="krati:target/test/producerspringtest"/>
</route>
```

Here is how the above example would look like using xml to define our route.

3.43.3.2. Example 2: Getting/Reading from a datastore

This example will show you how you can read the content of a datastore.

```
from("direct:get")
  .setHeader(KratiConstants.KRATI_OPERATION,
    constant(KratiConstants.KRATI_OPERATION_GET))
  .to("krati:target/test/producertest");
```

In the above example you can override any of the URI parameters with headers on the message.

Here is how the above example would look like using xml to define our route.

```
<route>
  <from uri="direct:get"/>
  <to uri="krati:target/test/producerspringtest?operation=CamelKratiGet"/>
</route>
```

3.43.3.3. Example 3: Consuming from a datastore

This example will consume all items that are under the specified datastore.

```
from("krati:target/test/consumertest")
```

```
.to("direct:next");
```

You can achieve the same goal by using xml, as you can see below.

```
from("krati:target/test/consumertest")
.to("direct:next");
```

3.43.4. Idempotent Repository

As already mentioned this component also offers an idempotent repository which can be used for filtering out duplicate messages.

```
from("direct://in").idempotentConsumer(header("messageId"),
new KratiIdempotentRepository("/tmp/idempotent").to("log://out");
```

3.44. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.44.1. 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&...`, where *option* can be:

Table 3.13.

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.

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).

3.44.2. Limitations

Currently camel-jsch only supports a [Producer](#) (i.e. copy files to another host).

3.45. LDAP

The ldap component allows you to perform searches in LDAP servers using filters as the message payload.

This component uses standard JNDI (`javax.naming` package) to access the server.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ldap</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.45.1. URI format

```
ldap:ldapServerBean[?options]
```

The *ldapServerBean* portion of the URI refers to a `DirContext` bean in the registry. The LDAP component only supports producer endpoints, which means that an ldap URI cannot appear in the `from` at the start of a route.

For more information about the `DirContext` bean, see its [JAVA API documentation](#) corresponding to the version of JAVA you are using.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.45.2. Options

Table 3.14.

Property	Default Value	Description
base	ou=system	The base DN for searches.
scope	subtree	Specifies how deeply to search the tree of entries, starting at the base DN. Value can be <code>object</code> , <code>onelevel</code> , or <code>subtree</code> .
pageSize	no paging used	When specified the ldap module uses paging to retrieve all results (most LDAP Servers throw an exception when trying to retrieve more than 1000 entries in one query). To be able to use this a <code>LdapContext</code> (subclass of <code>DirContext</code>) has to be passed in as <code>ldapServerBean</code> (otherwise an exception is thrown)
returnedAttributes	depends on LDAP Server (could be all or none)	Comma-separated list of attributes that should be set in each entry of the result

3.45.3. Result

The result is returned in the Out body as a `ArrayList<javax.naming.directory.SearchResult>` object.

3.45.4. DirContext

The URI, `ldap:ldapservice`, references a Spring bean with the ID, `ldapservice`. The `ldapservice` bean may be defined as follows:

```
<bean id="ldapservice" class="javax.naming.directory.InitialDirContext"
scope="prototype">
  <constructor-arg>
    <props>
      <prop>
        key="java.naming.factory.initial">com.sun.jndi.ldap.LdapCtxFactory</prop>
      <prop key="java.naming.provider.url">ldap://localhost:10389</prop>
      <prop key="java.naming.security.authentication">none</prop>
    </props>
  </constructor-arg>
</bean>
```

The preceding example declares a regular Sun based LDAP `DirContext` that connects anonymously to a locally hosted LDAP server.



DirContext objects are not required to support concurrency by contract. It is therefore important that the directory context is declared with the setting, `scope="prototype"`, in the bean definition or that the context supports concurrency. In the Spring framework, prototype scoped objects are instantiated each time they are looked up.

3.45.5. Samples

Following on from the Spring configuration above, the code sample below sends an LDAP request to filter search a group for a member. The Common Name is then extracted from the response.

```
ProducerTemplate<Exchange> template = exchange
    .getContext().createProducerTemplate();

Collection<?> results = (Collection<?>) (template
    .sendBody(
        "ldap:ldapservice?base=ou=mygroup,ou=groups,ou=system",
        "(member=uid=huntc,ou=users,ou=system)"));

if (results.size() > 0) {
    // Extract what we need from the device's profile

    Iterator<?> resultIter = results.iterator();
    SearchResult searchResult = (SearchResult) resultIter
        .next();
    Attributes attributes = searchResult
        .getAttributes();
    Attribute deviceCNAttr = attributes.get("cn");
    String deviceCN = (String) deviceCNAttr.get();

    ...
}
```

If no specific filter is required - for example, you just need to look up a single entry - specify a wildcard filter expression. For example, if the LDAP entry has a Common Name, use a filter expression like:

```
(cn=*)
```

3.45.5.1. Binding using credentials

A Camel end user donated this sample code he used to bind to the ldap server using credentials.

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");
props.setProperty(Context.PROVIDER_URL, "ldap://localhost:389");
props.setProperty(Context.URL_PKG_PREFIXES, "com.sun.jndi.url");
props.setProperty(Context.REFERRAL, "ignore");
props.setProperty(Context.SECURITY_AUTHENTICATION, "simple");
props.setProperty(Context.SECURITY_PRINCIPAL, "cn=Manager");
props.setProperty(Context.SECURITY_CREDENTIALS, "secret");

SimpleRegistry reg = new SimpleRegistry();
reg.put("myldap", new InitialLdapContext(props, null));

CamelContext context = new DefaultCamelContext(reg);
context.addRoutes(
    new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:start").to("ldap:myldap?base=ou=test");
        }
    }
);
context.start();

ProducerTemplate template = context.createProducerTemplate();

Endpoint endpoint = context.getEndpoint("direct:start");
Exchange exchange = endpoint.createExchange();
exchange.getIn().setBody("uid=test");
Exchange out = template.send(endpoint, exchange);

Collection<SearchResult> data = out.getOut().getBody(Collection.class);
assert data != null;
assert !data.isEmpty();

System.out.println(out.getOut().getBody());

context.stop();
```

3.46. 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.

3.46.1. URI format

```
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&...`



As of **Camel 2.12.4/2.13.1**, if there's single instance of `org.slf4j.Logger` found in the Registry, the **loggingCategory** is no longer used to create logger instance. The registered instance is used instead. Also it is possible to reference particular `Logger` instance using `?logger=#myLogger` URI parameter. Eventually, if there's no registered and URI logger parameter, the logger instance is created using **loggingCategory**.

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.



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 [Log](#).

And where *option* can be:

Table 3.15.

Option	Default	Type	Description
level	INFO	String	Logging level to use. Possible values: 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.
logger		logger	Camel 2.12.4/2.13.1: An optional reference to org.slf4j.Logger from Registry to use.

Note: `groupDelay` and `groupActiveOnly` are only applicable when using `groupInterval`

3.46.2. 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
skipBodyLineSeparator	true	Camel 2.12.2: Whether to skip line separators when logging the message body. This allows to log the message body in one line, setting this option to <code>false</code> will preserve any line separators from the body, which then will log the body <i>as is</i> .
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 <code>Exchange.EXCEPTION_CAUGHT</code>) and for instance a <code>doCatch</code> can catch exceptions. See Try Catch Finally .
showStackTrace	false	Show the stack trace, if an exchange has an exception. Only effective if one of <code>showAll</code> , <code>showException</code> or <code>showCaughtException</code> are enabled.
showFiles	false	Whether Camel should show file bodies or not (eg such as <code>java.io.File</code>).
showFuture	false	Whether Camel should show <code>java.util.concurrent.Future</code> bodies or not. If enabled Camel could potentially wait until the <code>Future</code> task is done. By default, this will not wait.
showStreams	false	Whether Camel should show stream bodies or not (eg such as <code>java.io.InputStream</code>). 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.

Starting with Camel 2.11, more advanced customization of the logging is possible, see the [Camel website](#) for more details.

3.46.3. 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>
```

3.46.4. 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");
```

3.46.5. 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");
```

3.46.6. 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"
```

3.46.7. Full customization of the logging output

Available as of Camel 2.11

With the options outlined in the [#Formatting](#) section, you can control much of the output of the logger. However, log lines will always follow this structure:

```
Exchange[Id:ID-machine-local-50656-1234567901234-1-2, ExchangePattern:InOut,
Properties:{CamelToEndpoint=log://org.apache.camel.component.log.TEST?showAll=true,
CamelCreatedTimestamp=Thu Mar 28 00:00:00 WET 2013},
Headers:{breadcrumbId=ID-machine-local-50656-1234567901234-1-1}, BodyType:String,
Body:Hello World, Out: null]
```

This format is unsuitable in some cases, perhaps because you need to...

- ... filter the headers and properties that are printed, to strike a balance between insight and verbosity.
- ... adjust the log message to whatever you deem most readable.
- ... tailor log messages for digestion by log mining systems, e.g. Splunk.
- ... print specific body types differently.
- ... etc.

Whenever you require absolute customization, you can create a class that implements the [ExchangeFormatter](#) interface. Within the `format(Exchange)` method you have access to the full `Exchange`, so you can select and extract the precise information you need, format it in a custom manner and return it. The return value will become the final log message.

You can have the Log component pick up your custom `ExchangeFormatter` in either of two ways:

Explicitly instantiating the LogComponent in your Registry:

```
<bean name="log" class="org.apache.camel.component.log.LogComponent">
  <property name="exchangeFormatter" ref="myCustomFormatter" />
</bean>
```

Convention over configuration:

Simply by registering a bean with the name `logFormatter`; the Log Component is intelligent enough to pick it up automatically.

```
<bean name="logFormatter" class="com.xyz.MyCustomExchangeFormatter" />
```

NOTE: the `ExchangeFormatter` gets applied to **all Log endpoints within that Camel Context**. If you need different `ExchangeFormatters` for different endpoints, just instantiate the `LogComponent` as many times as needed, and use the relevant bean name as the endpoint prefix.

From **Camel 2.11.2/2.12** onwards when using a custom log formatter, you can specify parameters in the log uri, which gets configured on the custom log formatter. Though when you do that you should define the "logFormatter" as prototype scoped so its not shared if you have different parameters, eg:

```
<bean name="logFormatter" class="com.xyz.MyCustomExchangeFormatter" scope="prototype"/>
```

And then we can have Camel routes using the log uri with different options:

```
<to uri="log:foo?param1=foo&param2=100"/>
...
<to uri="log:bar?param1=bar&param2=200"/>
```

3.46.7.1. Using Log component in OSGi

Improvement as of Camel 2.12.4/2.13.1

When using Log component inside OSGi (e.g., in Karaf), the underlying logging mechanisms are provided by PAX logging. It searches for a bundle which invokes `org.slf4j.LoggerFactory.getLogger()` method and associates the bundle with the logger instance. Without specifying custom `org.slf4j.Logger` instance, the logger created by Log component is associated with `camel-core` bundle.

In some scenarios it is required that the bundle associated with logger should be the bundle which contains route definition. To do this, either register single instance of `org.slf4j.Logger` in the Registry or reference it using `logger` URI parameter.

3.47. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.47.1. 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&...`

3.47.2. 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 <code>org.apache.lucene.analysis.Analyzer</code> . 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.

3.47.3. 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 <code>org.apache.lucene.analysis.Analyzer</code> . Lucene also offers a rich set of analyzers out of the box

Name	Default Value	Description
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

3.47.4. Sending/Receiving Messages to/from the cache

3.47.4.1. Message Headers

Header	Description
QUERY	The Lucene Query to performed on the index. The query may include wildcards and phrases

3.47.4.2. 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

3.47.4.3. Lucene Processor

There is a processor called LuceneQueryProcessor available to perform queries against lucene without the need to create a producer.

3.47.5. Lucene Usage Samples

3.47.5.1. Example: 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");
    }
}
```

```
};
```

3.47.5.2. Example: 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());
```

3.47.5.3. Example: 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");
    }
};
```

3.47.5.4. Example: 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");
}
};

```

3.48. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>

```

3.48.1. URI format and options

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&...`, where *option* can be:

Table 3.16.

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.)
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.
peek	true	Camel 2.11.3/2.12.2: Consumer only. Will mark the <code>javax.mail.Message</code> as peeked before processing the mail message. This applies to <code>IMAPMessage</code> messages types only. By using peek the mail will not be eager marked as <code>SEEN</code> on the mail server, which allows us to rollback the mail message if there is an error processing in Camel.
delete	false	Deletes the messages after they have been processed. This is done by setting the <code>DELETED</code> flag on the mail message. If false, the <code>SEEN</code> flag is set instead. You can override this configuration option by setting a header with the key <code>delete</code> to specify whether the mail should be deleted.
unseen	true	Is used to fetch only unseen messages (that is, new messages). Note that POP3 does not support the <code>SEEN</code> flag; use IMAP instead. Important: This option is not in use if you also use <code>searchTerm</code> options. Instead if you want to disable unseen when using <code>searchTerm</code> 's then add <code>searchTerm.unseen=false</code> as a term.
copyTo	null	Consumer only. After processing a mail message, it can be copied to a mail folder with the given name. You can override this configuration value with a header with the key <code>copyTo</code> , allowing you to copy messages to folder names configured at runtime.
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	CamelMailAlternativeBody	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-

Property	Default	Description
		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 <code>System.out</code> 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 <code>true</code> 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.
closeFolder	true	Whether the consumer should close the folder after polling. Setting this option to <code>false</code> while having <code>disconnect=false</code> will require the consumer to keep the folder open between polls.
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: <code>mail.pop3.forgettopheaders=true</code> . You can set multiple such options, for example: <code>mail.pop3.forgettopheaders=true&mail.mime.encodefilename=true</code> .
mapMailMessage	true	Specifies whether Camel should map the received mail message to Camel body/headers. If set to <code>true</code> , 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 <code>false</code> then the IN message contains a raw <code>javax.mail.Message</code> . You can retrieve this raw message by calling <code>exchange.getIn().getBody(javax.mail.Message.class)</code> .
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 <code>org.springframework.mail.javamail.JavaMailSender</code> instance in order to use a custom email implementation. If none provided, Camel uses the default, <code>org.springframework.mail.javamail.JavaMailSenderImpl</code> .
ignoreUnsupported-Charset	false	Option to let Camel ignore unsupported charset in the local JVM when sending mails. If the charset is unsupported then <code>charset=XXX</code> (where XXX represents the unsupported charset) is removed from the <code>content-type</code> and it relies on the platform default instead.
sslContext-Parameters	null	Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry . This reference overrides any configured <code>SSLContextParameters</code> at the component level. See Using the JSSE Configuration Utility for more information.
searchTerm	null	Starting with Camel 2.11, refers to a SearchTerm which allows for filtering mails based on search criteria such as subject, body, from, sent after a certain date, etc.
searchTerm.xxx	null	Starting with Camel 2.11, to configure search terms directly from the endpoint URI, which supports a limited number of terms defined by the SimpleSearchTerm class.

3.48.1.1. 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
```

3.48.1.2. 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

3.48.2. 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).

3.48.3. 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);
```

Starting with Camel 2.11, when using the MailProducer to send the mail to the server, you should be able to get the message id of the MimeMessage using the `CamelMailMessageId` key from the Camel message header.

3.48.4. 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);
```

3.48.5. 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.

3.48.6. 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 [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");
```

3.48.7. 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, see its [JAVA API documentation](#) corresponding to the version of JAVA you are using.

3.48.8. 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.

3.49. MINA 2

Available as of Camel 2.10

The **mina2**: component is a transport for working with [Apache MINA 2.x](#)

Favor using [Netty](#) as Netty is a much more active maintained and popular project than Apache Mina currently is

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mina2</artifactId>
```

```
<version>x.x.x</version>
<!-- use the same version as your Camel core version -->
</dependency>
```

3.49.1. URI format

```
mina2:tcp://hostname[:port][?options]
mina2:udp://hostname[:port][?options]
mina2:vm://hostname[:port][?options]
```

You can specify a codec in the [Registry](#) using the **codec** option. If you are using TCP and no codec is specified then the `textline` flag is used to determine if text line based codec or object serialization should be used instead. By default the object serialization is used.

For UDP if no codec is specified the default uses a basic `ByteBuffer` based codec.

The VM protocol is used as a direct forwarding mechanism in the same JVM.

A Mina producer has a default timeout value of 30 seconds, while it waits for a response from the remote server.

In normal use, `camel-mina` only supports marshalling the body content—message headers and exchange properties are not sent.

However, the option, **transferExchange**, does allow you to transfer the exchange itself over the wire. See options below.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.49.2. Options

Option	Default Value	Description
codec	null	You can refer to a named <code>ProtocolCodecFactory</code> instance in your Registry such as your <code>Spring ApplicationContext</code> , which is then used for the marshalling.
disconnect	false	Whether or not to <code>disconnect(close)</code> from Mina session right after use. Can be used for both consumer and producer.
textline	false	Only used for TCP. If no codec is specified, you can use this flag to indicate a text line based codec; if not specified or the value is <code>false</code> , then Object Serialization is assumed over TCP.
textlineDelimiter	DEFAULT	Only used for TCP and if textline=rowue . Sets the text line delimiter to use. Possible values are: <code>DEFAULT</code> , <code>AUTO</code> , <code>WINDOWS</code> , <code>UNIX</code> or <code>MAC</code> . If none provided, Camel will use <code>DEFAULT</code> . This delimiter is used to mark the end of text.
sync	rowue	Setting to set endpoint as one-way or request-response.
lazySessionCreation	rowue	Sessions can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started.
timeout	30000	You can configure the timeout that specifies how long to wait for a response from a remote server. The timeout unit is in milliseconds, so 60000 is 60 seconds. The timeout is only used for Mina producer.
encoding	<i>JVM Default</i>	You can configure the encoding (a charset name) to use for the TCP textline codec and the UDP protocol. If not provided, Camel will use the JVM default Charset .
rowansferExchange	false	Only used for TCP. You can rowansfer the exchange over the wire instead of just the body. The following fields are rowansferred: In body, Out body,

Option	Default Value	Description
		fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are <i>serializable</i> . Camel will exclude any non-serializable objects and log it at WARN level.
minaLogger	false	You can enable the Apache MINA logging filter. Apache MINA uses <code>slf4j</code> logging at INFO level to log all input and output.
filters	null	You can set a list of Mina IoFilters to register. The filters can be specified as a comma-separated list of bean references (e.g. <code>#filterBean1,#filterBean2</code>) where each bean must be of type <code>org.apache.mina.common.IoFilter</code> .
encoderMaxLineLength	-1	Set the textline protocol encoder max line length. By default the default value of Mina itself is used which are <code>Integer.MAX_VALUE</code> .
decoderMaxLineLength	-1	Set the textline protocol decoder max line length. By default the default value of Mina itself is used which are 1024.
maximumPoolSize	16	Number of worker threads in the worker pool for TCP and UDP (UDP requires Camel 2.11.3/2.12.2 onwards).
allowDefaultCodec	rowue	The mina component installs a default codec if both, <code>codec</code> is null and <code>textline</code> is false. Setting <code>allowDefaultCodec</code> to false prevents the mina component from installing a default codec as the first element in the filter chain. This is useful in scenarios where another filter must be the first in the filter chain, like the SSL filter.
disconnectOnNoReply	rowue	If sync is enabled then this option dictates <code>MinaConsumer</code> if it should disconnect where there is no reply to send back.
noReplyLogLevel	WARN	If sync is enabled this option dictates <code>MinaConsumer</code> which logging level to use when logging a there is no reply to send back. Values are: FATAL, ERROR, INFO, DEBUG, OFF.
orderedThreadPoolExecutor	rowue	Whether to use ordered thread pool, to ensure events are processed orderly on the same channel.
sslContextParameters	null	SSL configuration using an <code>org.apache.camel.util.jsse.SSLContextParameters</code> instance. See Using the JSSE Configuration Utility .
autoStartTls	rowue	Whether to auto start SSL handshake.

3.49.3. Using a custom codec

See the Mina how to write your own codec. To use your custom codec with `camel-mina`, you should register your codec in the [Registry](#); for example, by creating a bean in the Spring XML file. Then use the `codec` option to specify the bean ID of your codec. See [HL7](#) that has a custom codec.

3.49.4. Sample with sync=false

In this sample, Camel exposes a service that listens for TCP connections on port 6200. We use the **textline** codec. In our route, we create a Mina consumer endpoint that listens on port 6200:

```
from("mina2:tcp://localhost:" + port1 + "?textline=true&sync=false")
    .to("mock:result");
```

As the sample is part of a unit test, we test it by sending some data to it on port 6200.

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedBodiesReceived("Hello World");
```

```
template.sendBody("mina2:tcp://localhost:" + port1 + "?textline=true&sync=false",
"Hello World");

assertMockEndpointsSatisfied();
```

3.49.5. Sample with sync=true

In the next sample, we have a more common use case where we expose a TCP service on port 6201 also use the textline codec. However, this time we want to return a response, so we set the sync option to true on the consumer.

```
from("mina2:tcp://localhost:" + port2 + "?textline=true&sync=true").process(new
Processor() {
    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        exchange.getOut().setBody("Bye " + body);
    }
});
```

Then we test the sample by sending some data and retrieving the response using the `template.requestBody()` method. As we know the response is a `String`, we cast it to `String` and can assert that the response is, in fact, something we have dynamically set in our processor code logic.

```
String response = (String)template.requestBody("mina2:tcp://localhost:" + port2 +
"?textline=true&sync=true", "World");
assertEquals("Bye World", response);
```

3.49.6. Sample with Spring DSL

Spring DSL can, of course, also be used for [Mina](#). In the sample below we expose a TCP server on port 5555:

```
<route>
  <from uri="mina2:tcp://localhost:5555?textline=true"/>
  <to uri="bean:myTCPOrderHandler"/>
</route>
```

In the route above, we expose a TCP server on port 5555 using the textline codec. We let the Spring bean with ID, `myTCPOrderHandler`, handle the request and return a reply. For instance, the handler bean could be implemented as follows:

```
public String handleOrder(String payload) {
    ...
    return "Order: OK"
}
```

3.49.7. Closing Session When Complete

When acting as a server you sometimes want to close the session when, for example, a client conversion is finished. To instruct Camel to close the session, you should add a header with the key `CamelMinaCloseSessionWhenComplete` set to a boolean `true` value.

For instance, the example below will close the session after it has written the bye message back to the client:

```
from("mina2:tcp://localhost:8080?sync=true&textline=true").process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
```



```
exchange.getOut().setBody("Bye " + body);
exchange.getOut().setHeader(Mina2Constants.MINA_CLOSE_SESSION_WHEN_COMPLETE, true);
    }
});
```

3.49.8. Get the IoSession for message

You can get the IoSession from the message header with this key `Mina2Constants.MINA_IOSESSION`, and also get the local host address with the key `Mina2Constants.MINA_LOCAL_ADDRESS` and remote host address with the key `Mina2Constants.MINA_REMOTE_ADDRESS`.

3.49.9. Configuring Mina filters

Filters permit you to use some Mina Filters, such as `SslFilter`. You can also implement some customized filters. Please note that `codec` and `logger` are also implemented as Mina filters of type, `IoFilter`. Any filters you may define are appended to the end of the filter chain; that is, after `codec` and `logger`.

3.50. Mock

[Testing](#) of distributed and asynchronous processing is notoriously difficult. The [Mock](#), [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 [File](#) or [database](#), for example.



Remember that Mock is designed for testing; Mock endpoints keep received Exchanges in memory indefinitely. 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.

There are two options `retainFirst` and `retainLast` that can be used to limit the number of messages the Mock endpoints keep in memory.

3.50.1. 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&...`

3.50.2. Options

Option	Default	Description
reportGroup	null	A size to use a throughput logger for reporting
retainFirst	null	Camel 2.10: To only keep first X number of messages in memory.
retainLast	null	Camel 2.10: To only keep last X number of messages in memory.

3.50.3. 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.

3.50.3.1. 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();
```

3.50.4. 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 <code>JMSMessageID</code> if using JMS, or some unique reference number within the message.

Here's another example:

```
resultEndpoint.expectedBodiesReceived("firstMessageBody",
    "secondMessageBody", "thirdMessageBody");
```

3.50.4.1. 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](#).

3.50.5. Mocking existing endpoints

Camel now allows you to automatically mock existing endpoints in your Camel routes.



The endpoints are still in action, what happens is that a [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 designer
    // which has extended capabilities than the regular Route designer
    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 were 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  Advised endpoint [direct://foo] with mock endpoint [mock:direct:foo]
```



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 mock only certain endpoints using a pattern. For example to mock all `log` endpoints you can do as shown:

```
public void testAdvisedMockEndpointsWithPattern() throws Exception {
    // advice the first route using the inlined AdviceWith Route designer
```

```
// which has extended capabilities than the regular Route designer
context.getRouteDefinitions().get(0)
    .adviseWith(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");

// rest of code as previous example
...
}
```

The pattern supported can be a wildcard or a regular expression. See more details about this functionality on the [Apache Camel website](#).



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.

3.50.6. 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. There are 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.

3.50.7. Testing with arrival times

The *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 *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();
```



In the example above we use `seconds` as the time unit, but Camel offers `milliseconds`, and `minutes` as well.

3.51. MongoDB

Available as of Camel 2.10

According to Wikipedia: "NoSQL is a movement promoting a loosely defined class of non-relational data stores that break with a long history of relational databases and ACID guarantees." NoSQL solutions have grown in popularity in the last few years, and major extremely-used sites and services such as Facebook, LinkedIn, Twitter, etc. are known to use them extensively to achieve scalability and agility.

Basically, NoSQL solutions differ from traditional RDBMS (Relational Database Management Systems) in that they don't use SQL as their query language and generally don't offer ACID-like transactional behaviour nor relational data. Instead, they are designed around the concept of flexible data structures and schemas (meaning that the traditional concept of a database table with a fixed schema is dropped), extreme scalability on commodity hardware and blazing-fast processing.

MongoDB is a very popular NoSQL solution and the `camel-mongodb` component integrates Camel with MongoDB allowing you to interact with MongoDB collections both as a producer (performing operations on the collection) and as a consumer (consuming documents from a MongoDB collection).

MongoDB revolves around the concepts of documents (not as is office documents, but rather hierarchical data defined in JSON/BSON) and collections. This component page will assume you are familiar with them. Otherwise, visit <http://www.mongodb.org/>.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mongodb</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.51.1. URI format



```
mongodb:connectionBean?database=databaseName&collection=collectionName&operation=
operationName[&moreOptions...]
```

3.51.2. Endpoint options

MongoDB endpoints support the following options, depending on whether they are acting like a Producer or as a Consumer (options vary based on the consumer type too).

Name	Default Value	Description	Producer	Tailable Cursor Consumer
database	none	Required. The name of the database to which this endpoint will be bound. All operations will be executed against this database unless dynamicity is enabled and the <code>CamelMongoDbDatabase</code> header is set.	✓	✓
collection	none	Required. The name of the collection (within the specified database) to which this endpoint will be bound. All operations will be executed against this database unless dynamicity is enabled and the <code>CamelMongoDbDatabase</code> header is set.	✓	✓
collectionIndex	none	Camel 2.12: An optional single field index or compound index to create when inserting new collections.	✓	
operation	none	Required for producers. The id of the operation this endpoint will execute. Pick from the following: <ul style="list-style-type: none"> Query operations: <code>findById</code>, <code>findOneByQuery</code>, <code>findAll</code>, <code>count</code> Write operations: <code>insert</code>, <code>save</code>, <code>update</code> Delete operations: <code>remove</code> Other operations: <code>getDbStats</code>, <code>getColStats</code>, <code>aggregate</code> 	✓	
createCollection	true	Determines whether the collection will be automatically created in the MongoDB database during endpoint initialisation if it doesn't exist already. If this option is <code>false</code> and the collection doesn't exist, an initialisation exception will be thrown.	✓	
invokeGetLastError	false (behaviour may be inherited from connections <code>WriteConcern</code>)	Instructs the MongoDB Java driver to invoke <code>getLastError()</code> after every call. Default behaviour in version 2.7.2 of the MongoDB Java driver is that only network errors will cause the operation to fail, because the actual operation is executed asynchronously in the MongoDB server without holding up the client - to increase performance. The client can obtain the real result of the operation by explicitly invoking <code>getLastError()</code> on the <code>WriteResult</code> object returned or by setting the appropriate <code>WriteConcern</code> . If the backend operation has not finished yet, the client will block until the result is available. Setting this option to <code>true</code> will make the endpoint behave synchronously and return an <code>Exception</code> if the underlying operation failed.	✓	

Name	Default Value	Description	Producer	Tailable Cursor Consumer
writeConcern	none (driver's default)	Set a WriteConcern on the operation out of MongoDB's parameterised values. See WriteConcern.valueOf(String) .	✓	
writeConcernRef	none	Sets a custom WriteConcern that exists in the Registry. Specify the bean name.	✓	
readPreference	none	Sets a ReadPreference on the connection. Accepted values: the name of any inner subclass of ReadPreference . For example: PrimaryReadPreference, SecondaryReadPreference, TaggedReadPreference.	✓	
dynamicity	false	If set to true, the endpoint will inspect the CamelMongoDbDatabase and CamelMongoDbCollection headers of the incoming message, and if any of them exists, the target collection and/or database will be overridden for that particular operation. Set to false by default to avoid triggering the lookup on every Exchange if the feature is not desired.	✓	
writeResultAsHeader	false	Available as of Camel 2.10.3 and 2.11: In write operations (save, update, insert, etc.), instead of replacing the body with the WriteResult object returned by MongoDB, keep the input body untouched and place the WriteResult in the CamelMongoWriteResult header (constant <code>MongoDbConstants.WRITERESULT</code>).	✓	
persistentTail Tracking	false	Enables or disables persistent tail tracking for Tailable Cursor consumers. See below for more information.		✓
persistentId	none	Required if persistent tail tracking is enabled. The id of this persistent tail tracker, to separate its records from the rest on the tail-tracking collection.		✓
tailTracking IncreasingField	none	Required if persistent tail tracking is enabled. Correlation field in the incoming record which is of increasing nature and will be used to position the tailing cursor every time it is generated. The cursor will be (re)created with a query of type: <code>tailTrackIncreasingField > lastValue</code> (where <code>lastValue</code> is possibly recovered from persistent tail tracking). Can be of type Integer, Date, String, etc. NOTE: No support for dot notation at the current time, so the field should be at the top level of the document.		✓
cursorRegeneration Delay	1000ms	Establishes how long the endpoint will wait to regenerate the cursor after it has been killed by the MongoDB server (normal behaviour).		✓
tailTrackDb	same as endpoint's	Database on which the persistent tail tracker will store its runtime information.		✓

Name	Default Value	Description	Producer	Tailable Cursor Consumer
tailTrackCollection	camelTailTracking	Collection on which the persistent tail tracker will store its runtime information.		
tailTrackField	lastTrackingValue	Field in which the persistent tail tracker will store the last tracked value.		

3.51.3. MongoDB operations - producer endpoints

3.51.3.1. Query operations

findById

This operation retrieves only one element from the collection whose `_id` field matches the content of the IN message body. The incoming object can be anything that has an equivalent to a BSON type. See <http://bsonspec.org/#/specification> and <http://www.mongodb.org/display/DOCS/Java+Types>.

```
from("direct:findById")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=operation=findById")
    .to("mock:resultFindById");
```

This operation supports specifying a fields filter. See [Specifying a fields filter](#).

findOneByQuery

Use this operation to retrieve just one element from the collection that matches a MongoDB query. **The query object is extracted from the IN message body**, i.e. it should be of type `DBObject` or convertible to `DBObject`. It can be a JSON String or a Hashmap. See [#Type conversions](#) for more info.

Example with no query (returns any object of the collection):

```
from("direct:findOneByQuery")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=operation=findOneByQuery")
    .to("mock:resultFindOneByQuery");
```

Example with a query (returns one matching result):

```
from("direct:findOneByQuery")
    .setBody().constant("{ \"name\": \"Raul Kripalani\" }")
    .to("mongodb:myDb?database=flights&collection=tickets&operation=operation=findOneByQuery")
    .to("mock:resultFindOneByQuery");
```

This operation supports specifying a fields filter. See [Specifying a fields filter](#).

findAll

The `findAll` operation returns all documents matching a query, or none at all, in which case all documents contained in the collection are returned. **The query object is extracted from the IN message body**, i.e. it should

be of type `DBObject` or convertible to `DBObject`. It can be a JSON String or a Hashmap. See [#Type conversions](#) for more info.

Example with no query (returns all object in the collection):

```
from("direct:findAll")
  .to("mongodb:myDb?database=flights&collection=tickets&operation=findAll")
  .to("mock:resultFindAll");
```

Example with a query (returns all matching results):

```
from("direct:findAll")
  .setBody().constant("{ \"name\": \"Raul Kripalani\" }")
  .to("mongodb:myDb?database=flights&collection=tickets&operation=findAll")
  .to("mock:resultFindAll");
```

Paging and efficient retrieval is supported via the following headers:

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDb NumToSkip	MongoDbConstants .NUM_TO_SKIP	Discards a given number of elements at the beginning of the cursor.	int/Integer
CamelMongoDb Limit	MongoDbConstants .LIMIT	Limits the number of elements returned.	int/Integer
CamelMongoDb BatchSize	MongoDbConstants .BATCH_SIZE	Limits the number of elements returned in one batch. A cursor typically fetches a batch of result objects and store them locally. If batchSize is positive, it represents the size of each batch of objects retrieved. It can be adjusted to optimize performance and limit data transfer. If batchSize is negative, it will limit of number objects returned, that fit within the max batch size limit (usually 4MB), and cursor will be closed. For example if batchSize is -10, then the server will return a maximum of 10 documents and as many as can fit in 4MB, then close the cursor. Note that this feature is different from limit() in that documents must fit within a maximum size, and it removes the need to send a request to close the cursor server-side. The batch size can be changed even after a cursor is iterated, in which case the setting will apply on the next batch retrieval.	int/Integer

Additionally, you can set a `sortBy` criteria by putting the relevant `DBObject` describing your sorting in the `CamelMongoDbSortBy` header, quick constant: `MongoDbConstants.SORT_BY`.

The `findAll` operation will also return the following OUT headers to enable you to iterate through result pages if you are using paging:

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDb ResultTotalSize	MongoDbConstants .RESULT_TOTAL_SIZE	Number of objects matching the query. This does not take limit/skip into consideration.	int/Integer
CamelMongoDb ResultPageSize	MongoDbConstants .RESULT_PAGE_SIZE	Number of objects matching the query. This does not take limit/skip into consideration.	int/Integer

This operation supports specifying a fields filter. See [Specifying a fields filter](#).

Specifying a fields filter

Query operations will, by default, return the matching objects in their entirety (with all their fields). If your documents are large and you only require retrieving a subset of their fields, you can specify a field filter in all query operations, simply by setting the relevant `DBObject` (or type convertible to `DBObject`, such as a JSON String, Map, etc.) on the `CamelMongoDbFieldsFilter` header, constant shortcut: `MongoDbConstants.FIELDS_FILTER`.

Here is an example that uses MongoDB's `BasicDBObjectBuilder` to simplify the creation of `DBObject`s. It retrieves all fields except `_id` and `boringField`:

```
// route:
from("direct:findAll").to("mongodb:myDb?database=flights&collection=tickets&operation=findAll")
DBObject fieldFilter = BasicDBObjectBuilder.start().add("_id", 0).add("boringField", 0).get();
Object result = template.requestBodyAndHeader("direct:findAll", (Object) null,
MongoDbConstants.FIELDS_FILTER, fieldFilter);
```

3.51.3.2. Create/update operations

insert

Inserts an new object into the MongoDB collection, taken from the IN message body. Type conversion is attempted to turn it into `DBObject` or a `List`.

Two modes are supported: single insert and multiple insert. For multiple insert, the endpoint will expect a `List`, `Array` or `Collections` of objects of any type, as long as they are - or can be converted to - `DBObject`. All objects are inserted at once. The endpoint will intelligently decide which backend operation to invoke (single or multiple insert) depending on the input.

Example:

```
from("direct:insert")
.to("mongodb:myDb?database=flights&collection=tickets&operation=insert");
```

The operation will return a `WriteResult`, and depending on the `WriteConcern` or the value of the `invokeGetLastError` option, `getLastError()` would have been called already or not. If you want to access the ultimate result of the write operation, you need to retrieve the `CommandResult` by calling `getLastError()` or `getCachedLastError()` on the `WriteResult`. Then you can verify the result by calling `CommandResult.ok()`, `CommandResult.getErrorMessage()` and/or `CommandResult.getException()`.

Note that the new object's `_id` must be unique in the collection. If you don't specify the value, MongoDB will automatically generate one for you. But if you do specify it and it is not unique, the insert operation will fail (and for Camel to notice, you will need to enable `invokeGetLastError` or set a `WriteConcern` that waits for the write result).

This is not a limitation of the component, but it is how things work in MongoDB for higher throughput. If you are using a custom `_id`, you are expected to ensure at the application level that is unique (and this is a good practice too).

save

The save operation is equivalent to an *upsert* (UPDATE, inSERT) operation, where the record will be updated, and if it doesn't exist, it will be inserted, all in one atomic operation. MongoDB will perform the matching based on the `_id` field.

Beware that in case of an update, the object is replaced entirely and the usage of [MongoDB's \\$modifiers](#) is not permitted. Therefore, if you want to manipulate the object if it already exists, you have two options:

1. perform a query to retrieve the entire object first along with all its fields (may not be efficient), alter it inside Camel and then save it.
2. use the update operation with [\\$modifiers](#), which will execute the update at the server-side instead. You can enable the upsert flag, in which case if an insert is required, MongoDB will apply the \$modifiers to the filter query object and insert the result.

For example:

```
from("direct:insert")
  .to("mongodb:myDb?database=flights&collection=tickets&operation=save");
```

update

Update one or multiple records on the collection. Requires a List<DBObject> as the IN message body containing exactly 2 elements:

- Element 1 (index 0) => filter query => determines what objects will be affected, same as a typical query object
- Element 2 (index 1) => update rules => how matched objects will be updated. All [modifier operations](#) from MongoDB are supported.

By default, MongoDB will only update 1 object even if multiple objects match the filter query. To instruct MongoDB to update **all** matching records, set the CamelMongoDbMultiUpdate IN message header to true.

A header with key CamelMongoDbRecordsAffected will be returned (MongoDbConstants.RECORDS_AFFECTED constant) with the number of records updated (copied from WriteResult.getN()).

Supports the following IN message headers:

Header key	Quick constant	Description (extracted from MongoDB API doc)	Expected type
CamelMongoDbMultiUpdate	MongoDbConstants .MULTIUPDATE	If the update should be applied to all objects matching. See http://www.mongodb.org/display/DOCS/Atomic+Operations	boolean/Boolean
CamelMongoDbUpsert	MongoDbConstants .UPSERT	If the database should create the element if it does not exist	boolean/Boolean

For example, the following will update **all** records whose filterField field equals true by setting the value of the "scientist" field to "Darwin":

```
// route:
from("direct:update").to("mongodb:myDb?database=science&collection=notableScientist
s&operation=update");
DBObject filterField = new BasicDBObject("filterField", true);
DBObject updateObj = new BasicDBObject("$set", new BasicDBObject("scientist",
"Darwin"));
Object result = template.requestBodyAndHeader("direct:update", new Object[]
{filterField, updateObj}, MongoDbConstants.MULTIUPDATE, true);
```

3.51.3.3. Delete operations

remove

Remove matching records from the collection. The IN message body will act as the removal filter query, and is expected to be of type DBObject or a type convertible to it.

The following example will remove all objects whose field 'conditionField' equals true, in the science database, notableScientists collection:

```
// route:
from("direct:remove").to("mongodb:myDb?database=science&collection=notableScientist
s&operation=remove");
DBObject conditionField = new BasicDBObject("conditionField", true);
Object result = template.requestBody("direct:remove", conditionField);
```

A header with key CamelMongoDbRecordsAffected is returned (MongoDbConstants.RECORDS_AFFECTED constant) with type int, containing the number of records deleted (copied from WriteResult.getN()).

3.51.3.4. Other operations

count

Returns the total number of objects in a collection, returning a Long as the OUT message body.

The following example will count the number of records in the "dynamicCollectionName" collection. Notice how dynamicity is enabled, and as a result, the operation will not run against the "notableScientists" collection, but against the "dynamicCollectionName" collection.

```
// from("direct:count").to("mongodb:myDb?database=tickets&collection=flights&operation
=count&dynamicity=true");
Long result = template.requestBodyAndHeader("direct:count", "irrelevantBody",
MongoDbConstants.COLLECTION, "dynamicCollectionName");
assertTrue("Result is not of type Long", result instanceof Long);
```

getDbStats

Equivalent of running the db.stats() command in the MongoDB shell, which displays useful statistic figures about the database.

For example:

```
> db.stats();
{
  "db" : "test",
  "collections" : 7,
  "objects" : 719,
  "avgObjSize" : 59.73296244784423,
  "dataSize" : 42948,
  "storageSize" : 1000058880,
  "numExtents" : 9,
  "indexes" : 4,
  "indexSize" : 32704,
  "fileSize" : 1275068416,
  "nsSizeMB" : 16,
  "ok" : 1
}
```

Usage example:

```
//
from("direct:getDbStats").to("mongodb:myDb?database=flights&collection=tickets&oper
ation=getDbStats");
```

```
Object result = template.requestBody("direct:getDbStats", "irrelevantBody");
assertTrue("Result is not of type DBObject", result instanceof DBObject);
```

The operation will return a data structure similar to the one displayed in the shell, in the form of a `DBObject` in the OUT message body.

getColStats

Equivalent of running the `db.collection.stats()` command in the MongoDB shell, which displays useful statistic figures about the collection. For example:

For example:

```
> db.camelTest.stats();
{
  "ns" : "test.camelTest",
  "count" : 100,
  "size" : 5792,
  "avgObjSize" : 57.92,
  "storageSize" : 20480,
  "numExtents" : 2,
  "nindexes" : 1,
  "lastExtentSize" : 16384,
  "paddingFactor" : 1,
  "flags" : 1,
  "totalIndexSize" : 8176,
  "indexSizes" : {
    "_id_" : 8176
  },
  "ok" : 1
}
```

Usage example:

```
//
from("direct:getColStats").to("mongodb:myDb?database=flights&collection=tickets&operation=getColStats");
Object result = template.requestBody("direct:getColStats", "irrelevantBody");
assertTrue("Result is not of type DBObject", result instanceof DBObject);
```

The operation will return a data structure similar to the one displayed in the shell, in the form of a `DBObject` in the OUT message body.

3.51.3.5. Dynamic operations

An Exchange can override the endpoint's fixed operation by setting the `CamelMongoDbOperation` header, defined by the `MongoDbConstants.OPERATION_HEADER` constant.

The values supported are determined by the `MongoDbOperation` enumeration and match the accepted values for the `operation` parameter on the endpoint URI.

For example:

```
//
from("direct:insert").to("mongodb:myDb?database=flights&collection=tickets&operation=insert");
Object result = template.requestBodyAndHeader("direct:insert", "irrelevantBody",
MongoDbConstants.OPERATION_HEADER, "count");
```

```
assertTrue("Result is not of type Long", result instanceof Long);
```

3.51.4. Tailable Cursor Consumer

MongoDB offers a mechanism to instantaneously consume ongoing data from a collection, by keeping the cursor open just like the `tail -f` command of *nix systems. This mechanism is significantly more efficient than a scheduled poll, due to the fact that the server pushes new data to the client as it becomes available, rather than making the client ping back at scheduled intervals to fetch new data. It also reduces otherwise redundant network traffic.

There is only one requisite to use tailable cursors: the collection must be a "capped collection", meaning that it will only hold N objects, and when the limit is reached, MongoDB flushes old objects in the same order they were originally inserted. For more information, please refer to: <http://www.mongodb.org/display/DOCS/Tailable+Cursors>.

The Camel MongoDB component implements a tailable cursor consumer, making this feature available for you to use in your Camel routes. As new objects are inserted, MongoDB will push them as `DBObject`s in natural order to your tailable cursor consumer, who will transform them to an `Exchange` and will trigger your route logic.

3.51.4.1. How the tailable cursor consumer works

To turn a cursor into a tailable cursor, a few special flags are to be signalled to MongoDB when first generating the cursor. Once created, the cursor will then stay open and will block upon calling the `DBCursor.next()` method until new data arrives. However, the MongoDB server reserves itself the right to kill your cursor if new data doesn't appear after an indeterminate period. If you are interested to continue consuming new data, you have to regenerate the cursor. And to do so, you will have to remember the position where you left off or else you will start consuming from the top again.

The Camel MongoDB tailable cursor consumer takes care of all these tasks for you. You will just need to provide the key to some field in your data of increasing nature, which will act as a marker to position your cursor every time it is regenerated, e.g. a timestamp, a sequential ID, etc. It can be of any datatype supported by MongoDB. Date, Strings and Integers are found to work well. We call this mechanism "tail tracking" in the context of this component.

The consumer will remember the last value of this field and whenever the cursor is to be regenerated, it will run the query with a filter like: `increasingField > lastValue`, so that only unread data is consumed.

Setting the increasing field: Set the key of the increasing field on the endpoint URI `tailTrackingIncreasingField` option. In Camel 2.10, it must be a top-level field in your data, as nested navigation for this field is not yet supported. That is, the "timestamp" field is okay, but "nested.timestamp" will not work. Please open a ticket in the Camel JIRA if you do require support for nested increasing fields.

Cursor regeneration delay: One thing to note is that if new data is not already available upon initialisation, MongoDB will kill the cursor instantly. Since we don't want to overwhelm the server in this case, a `cursorRegenerationDelay` option has been introduced (with a default value of 1000ms.), which you can modify to suit your needs.

An example:

```
from("mongodb:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime")
    .id("tailableCursorConsumer1")
    .autoStartup(false)
    .to("mock:test");
```

The above route will consume from the "flights.cancellations" capped collection, using "departureTime" as the increasing field, with a default regeneration cursor delay of 1000ms.

3.51.4.2. Persistent tail tracking

Standard tail tracking is volatile and the last value is only kept in memory. However, in practice you will need to restart your Camel container every now and then, but your last value would then be lost and your tailable cursor consumer would start consuming from the top again, very likely sending duplicate records into your route.

To overcome this situation, you can enable the **persistent tail tracking** feature to keep track of the last consumed increasing value in a special collection inside your MongoDB database too. When the consumer initialises again, it will restore the last tracked value and continue as if nothing happened.

The last read value is persisted on two occasions: every time the cursor is regenerated and when the consumer shuts down. We may consider persisting at regular intervals too in the future (flush every 5 seconds) for added robustness if the demand is there. To request this feature, please open a ticket in the Camel JIRA.

3.51.4.3. Enabling persistent tail tracking

To enable this function, set at least the following options on the endpoint URI:

- `persistentTailTracking` option to `true`
- `persistentId` option to a unique identifier for this consumer, so that the same collection can be reused across many consumers

Additionally, you can set the `tailTrackDb`, `tailTrackCollection` and `tailTrackField` options to customise where the runtime information will be stored. Refer to the endpoint options table at the top of this page for descriptions of each option.

For example, the following route will consume from the "flights.cancellations" capped collection, using "departureTime" as the increasing field, with a default regeneration cursor delay of 1000ms, with persistent tail tracking turned on, and persisting under the "cancellationsTracker" id on the "flights.camelTailTracking", storing the last processed value under the "lastTrackingValue" field (`camelTailTracking` and `lastTrackingValue` are defaults).

```
from("mongodb:myDb?database=flights&collection=cancellations
&tailTrackIncreasingField=departureTime&persistentTailTracking=true" +
    "&persistentId=cancellationsTracker")
    .id("tailableCursorConsumer2")
    .autoStartup(false)
    .to("mock:test");
```

Below is another example identical to the one above, but where the persistent tail tracking runtime information will be stored under the "trackers.camelTrackers" collection, in the "lastProcessedDepartureTime" field:

```
from("mongodb:myDb?
database=flights&collection=cancellations&tailTrackIncreasingField=departureTime
&persistentTailTracking=true" +
    "&persistentId=cancellationsTracker&tailTrackDb=trackers&tailTrackCollection=
camelTrackers" +
    "&tailTrackField=lastProcessedDepartureTime")
    .id("tailableCursorConsumer3")
    .autoStartup(false)
    .to("mock:test");
```

3.51.5. Type conversions

The `MongoDbBasicConverters` type converter included with the camel-mongodb component provides the following conversions:

Name	From type	To type	How?
fromMapToDBObject	Map	DBObject	constructs a new BasicDBObject via the new BasicDBObject(Map m) constructor
fromBasicDBObjectToMap	BasicDBObject	Map	BasicDBObject already implements Map
fromStringToDBObject	String	DBObject	uses com.mongodb.util. JSON.parse(Strings)
fromAnyObjectToDBObject	Object	DBObject	uses the Jackson library to convert the object to a Map, which is in turn used to initialise a new BasicDBObject

This type converter is auto-discovered, so you don't need to configure anything manually.

3.52. MQTT

Available as of Camel 2.10

The **mqtt** component is used for communicating with **MQTT** compliant message brokers, like [Apache ActiveMQ](#) or [Mosquitto](#)

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-mqtt</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.52.1. URI format

```
mqtt://name[?options]
```

Where **name** is the name you want to assign the component.

3.52.2. Options

Property	Default	Description
host	tcp://127.0.0.1:1883	The URI of the MQTT broker to connect too - this component also supports SSL - e.g. <code>ssl://127.0.0.1:8883</code>
localAddress		The local InetAddress and port to use
connectAttemptsMax	-1	The maximum number of attempts to establish an initial connection - -1 in infinite.
reconnectAttemptsMax	-1	The maximum number of attempts to re-establish a connection after failure - -1 in infinite.

Property	Default	Description
reconnectDelay	10	The time in milliseconds between attempts to reestablish an initial or failed connection
reconnectBackOffMultiplier	2.0	The multiplier to use to the delay between connection attempts for successive failed connection attempts
reconnectDelayMax	30000	The maximum time in milliseconds between a new attempt to establish a connection. So even using the reconnectBackOffMultiplier, this property will define the maximum delay before another connection attempt to the MQTT broker
QoS	AtLeastOnce	The MQTT Quality of Service to use for message exchanges. It can be one of AtMostOnce , AtLeastOnce or ExactlyOnce
subscribeTopicName		The name of the Topic to subscribe to for messages
publishTopicName	camel/mqtt/test	The default Topic to publish messages on
byDefaultRetain	false	The default retain policy to be used on messages sent to the MQTT broker
mqttTopicPropertyName	_MQTTTopicPropertyName+	The property name to look for on an Exchange for an individual published message. If this is set - the name will be used as the Topic to publish a message to the MQTT message broker.
mqttRetainPropertyName	MQTTRetain	The property name to look for on an Exchange for an individual published message. If this is set (expects a Boolean value) - then the retain property will be set on the message sent to the MQTT message broker.
mqttQosPropertyName	MQTTQos	The property name to look for on an Exchange for an individual published message. If this is set (one of AtMostOnce , AtLeastOnce or ExactlyOnce) - then that QoS will be set on the message sent to the MQTT message broker.
connectWaitInSeconds	10	Delay in seconds the Component will wait for a connection to be established to the MQTT broker
disconnectWaitInSeconds	5	the number of seconds the Component will wait for a valid disconnect on stop() from the MQTT broker
sendWaitInSeconds	5	The maximum time the Component will wait for a receipt from the MQTT broker to acknowledge a published message before throwing an exception

You can append query options to the URI in the following format, ?option=value&option=value&...

3.52.3. Samples

Sending messages:

```
from("direct:foo").to("mqtt:cheese?publishTopicName=test.mqtt.topic");
```

Consuming messages:

```
from("mqtt:bar?subscribeTopicName=test.mqtt.topic").transform(body().convertToString()).to("mock:result")
```

3.52.4. Endpoints

Camel supports the [Message Endpoint](#) pattern using the [Endpoint](#) interface. Endpoints are usually created by a [Component](#) and Endpoints are usually referred to in the [DSL](#) via their [URIs](#).

From an Endpoint you can use the following methods

- [createProducer\(\)](#) will create a [Producer](#) for sending message exchanges to the endpoint
- [createConsumer\(\)](#) implements the [Event Driven Consumer](#) pattern for consuming message exchanges from the endpoint via a [Processor](#) when creating a [Consumer](#)
- [createPollingConsumer\(\)](#) implements the [Polling Consumer](#) pattern for consuming message exchanges from the endpoint via a [PollingConsumer](#)

3.53. Mustache

Available as of Camel 2.12

The **mustache:** component allows for processing a message using a [Mustache](#) 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-mustache</artifactId>
<version>x.x.x</version> <!-- use the same version as your Camel core version -->
</dependency>
```

3.53.1. URI format

```
mustache:templateName[?options]
```

Where **templateName** is the classpath-local URI of the template to invoke; or the complete URL of the remote template (eg: `file://folder/myfile.mustache`).

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.53.2. Options

Name	Required	Description
encoding	null	Character encoding of the resource content.

Name	Required	Description
startDelimiter	{{	Characters used to mark template code beginning.
endDelimiter	}}	Characters used to mark template code end.

3.53.3. Mustache Context

Camel will provide exchange information in the Mustache 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).

3.53.4. 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	Support Version
MustacheConstants.MUSTACHE_RESOURCE_URI	String	A URI for the template resource to use instead of the endpoint configured.	
MustacheConstants.MUSTACHE_TEMPLATE	String	The template to use instead of the endpoint configured.	

3.53.5. Samples

For example you could use something like:

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

To use a Mustache 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("mustache:com/acme/MyResponse.mustache").
to("activemq:Another.Queue");
```

It's possible to specify what template the component should use dynamically via a header, so for example:

```
from("direct:in").
setHeader(MustacheConstants.MUSTACHE_RESOURCE_URI).constant("path/to/my/template.mu
```

```
stache").
to("mustache:dummy");
```

3.53.6. The Email Sample

In this sample we want to use Mustache templating for an order confirmation email. The email template is laid out in Mustache as:

```
Dear {{headers.lastName}}, {{headers.firstName}}

Thanks for the order of {{headers.item}}.

Regards Camel Riders Bookstore
{{body}}
```

3.54. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.54.1. URI format and Options

```
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&...`, where *option* can be:

Table 3.17.

Option	Type	Default	Description
<code>consumer.onConsume</code>	String	null	Statements to run after consuming. Can be used, for example, to update rows after they have been consumed and processed in Camel. Multiple statements can be separated with commas.
<code>consumer.useIterator</code>	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.
<code>consumer.routeEmptyResultSet</code>	boolean	false	Sets whether empty result sets should be routed.
<code>statementType</code>	StatementType	null	Mandatory to specify for the Producer to control which kind of operation to invoke. The enum values are: <code>SelectOne</code> , <code>SelectList</code> , <code>Insert</code> ,

Option	Type	Default	Description
			InsertList, Update, UpdateList, Delete, DeleteList.
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.

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.

3.54.2. 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.

3.54.3. 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.

3.54.4. 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>
```

3.54.5. 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");
```

3.54.5.1. 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>
```

3.55. Netty HTTP

Available as of Camel 2.12

The **netty-http** component is an extension to [Netty](#) component to facilitate HTTP transport with [Netty](#).

This camel component supports both producer and consumer endpoints.

Upgrade to Netty 4.0 planne

This component is intended to be upgraded to use Netty 4.0 when `camel-netty4` component has finished being upgraded. At the time being this component is still based on Netty 3.x. The upgrade is intended to be as backwards compatible as possible.

Stream

Netty 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 re-read multiple times.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty-http</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.55.1. URI format

The URI scheme for a netty component is as follows

```
netty-http:http://localhost:8080[?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`



You may be wondering how Camel recognizes URI query parameters and endpoint options. For example you might create endpoint URI as follows - `netty-http:http://example.com?myParam=myValue&compression=true`. In this example `myParam` is the HTTP parameter, while `compression` is the Camel endpoint option. The strategy used by Camel in such situations is to resolve available endpoint options and remove them from the URI. It means that for the discussed example, the HTTP request sent by Netty HTTP producer to the endpoint will look as follows - `http://example.com?myParam=myValue`, because `compression` endpoint option will be resolved and removed from the target URL.

3.55.2. HTTP Options

A lot more options

Important: This component inherits all the options from [Netty](#). So make sure to look at the [Netty](#) documentation as well.

Notice that some options from [Netty](#) is not applicable when using this [Netty HTTP](#) component, such as options related to UDP transport.

Name	Default Value	Description
<code>chunkedMaxContentLength</code>	<code>1mb</code>	Value in bytes the max content length per chunked frame received on the Netty HTTP server.
<code>compression</code>	<code>false</code>	Allow using <code>gzip/deflate</code> for compression on the Netty HTTP server if the client supports it from the HTTP headers.
<code>headerFilterStrategy</code>		To use a custom <code>org.apache.camel.spi.HeaderFilterStrategy</code> to filter headers.

Name	Default Value	Description
httpMethodRestrict		To disable HTTP methods on the Netty HTTP consumer. You can specify multiple separated by comma.
mapHeaders	true	<p>If this option is enabled, then during binding from Netty to Camel Message then the headers will be mapped as well (eg added as header to the Camel Message as well). You can turn off this option to disable this. The headers can still be accessed from the <code>org.apache.camel.component.netty.http.Netty</code></p> <p><code>HttpMessage</code> message with the method <code>getHttpRequest()</code> that returns the Netty HTTP request <code>org.jboss.netty.handler.codec.http.HttpRequest</code> instance.</p>
matchOnUriPrefix	false	Whether or not Camel should try to find a target consumer by matching the URI prefix if no exact match is found. See further below for more details.
nettyHttpBinding		To use a custom <code>org.apache.camel.component.netty.http.Netty</code> <code>HttpBinding</code> for binding to/from Netty and Camel Message API.
bridgeEndpoint	false	If the option is <code>true</code> , the producer will ignore the <code>Exchange.HTTP_URI</code> header, and use the endpoint's URI for request. You may also set the <code>throwExceptionOnFailure</code> to be <code>false</code> to let the producer send all the fault response back.
throwExceptionOnFailure	true	Option to disable throwing the <code>HttpOperationFailedException</code> in case of failed responses from the remote server. This allows you to get all responses regardless of the HTTP status code.
traceEnabled	false	Specifies whether to enable HTTP TRACE for this Netty HTTP consumer. By default TRACE is turned off.
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 <code>application/x-java-serialized-object</code> content type. On the producer side the exception will be deserialized and thrown as is, instead of the <code>HttpOperationFailedException</code> . The caused exception is required to be serialized.
urlDecodeHeaders		<p>If this option is enabled, then during binding from Netty to Camel Message then the header values will be URL decoded (eg <code>%20</code> will be a space character. Notice this option is used by the default <code>org.apache.camel.component.netty.http</code>.</p> <p><code>NettyHttpBinding</code> and therefore if you implement a custom <code>org.apache.camel.component.netty</code>.</p> <p><code>http.NettyHttpBinding</code> then you would need to decode the headers accordingly to this option. Notice: This option is default <code>true</code> for Camel 2.12.x, and default <code>false</code> from Camel 2.13 onwards.</p>
nettySharedHttpServer	null	To use a shared Netty HTTP server. See Netty HTTP Server Example for more details.
disableStreamCache	false	Determines whether or not the raw input stream from Netty <code>HttpRequest#getContent()</code> is cached or not (Camel will read the stream into a in light-weight memory based Stream caching) cache. By default Camel will cache the Netty 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 <code>true</code> when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. Mind that if you enable this option, then you cannot read the Netty stream multiple times out of the box, and you would need manually to reset the reader index on the Netty raw stream.
securityConfiguration	null	<p>Consumer only. Refers to a <code>org.apache.camel.component.netty.http.Netty</code></p> <p><code>HttpSecurityConfiguration</code> for configuring secure web resources.</p>
send503whenSuspended	true	Consumer only. Whether to send back HTTP status code 503 when the consumer has been suspended. If the option is <code>false</code> then the Netty Acceptor is unbound when the consumer is suspended, so clients cannot connect anymore.

The `NettyHttpSecurityConfiguration` has the following options:

Name	Default Value	Description
authenticate	true	Whether authentication is enabled. Can be used to quickly turn this off.
constraint	Basic	The constraint supported. Currently only Basic is implemented and supported.
realm	null	The name of the JAAS security realm. This option is mandatory.
securityConstraint	null	Allows to plugin a security constraint mapper where you can define ACL to web resources.
securityAuthenticator	null	Allows to plugin a authenticator that performs the authentication. If none has been configured then the <code>org.apache.camel.component.netty.http.JAAS SecurityAuthenticator</code> is used by default.
loginDeniedLoggingLevel	DEBUG	Logging level used when a login attempt failed, which allows to see more details why the login failed.
roleClassName	null	To specify FQN class names of <code>Principal</code> implementations that contains user roles. If none has been specified, then the Netty HTTP component will by default assume a <code>Principal</code> is role based if its FQN classname has the lower-case word <code>role</code> in its classname. You can specify multiple class names separated by comma.

3.55.3. Message Headers

The following headers can be used on the producer to control the HTTP request.

Name	Type	Description
CamelHttpMethod	String	Allow to control what HTTP method to use such as GET, POST, TRACE etc. The type can also be a <code>org.jboss.netty.handler.codec.http.HttpMethod</code> instance.
CamelHttpQuery	String	Allows to provide URI query parameters as a <code>String</code> value that overrides the endpoint configuration. Separate multiple parameters using the <code>&</code> sign. For example: <code>foo=bar&beer=yes</code> .
CamelHttpPath	String	Camel 2.13.1/2.12.4: Allows to provide URI context-path and query parameters as a <code>String</code> value that overrides the endpoint configuration. This allows to reuse the same producer for calling same remote http server, but using a dynamic context-path and query parameters.
Content-Type	String	To set the content-type of the HTTP body. For example: <code>text/plain; charset="UTF-8"</code> .

The following headers is provided as meta-data when a route starts from an [Netty HTTP](#) endpoint:

The description in the table takes offset in a route having: `from("netty-http:http:0.0.0.0:8080/myapp")...`

Name	Type	Description
CamelHttpMethod	String	The HTTP method used, such as GET, POST, <code>rowACE</code> etc.
CamelHttpUrl	String	The URL including protocol, host and port, etc: <code>http://0.0.0.0:8080/myapp</code>
CamelHttpUri	String	The URI without protocol, host and port, etc: <code>/myapp</code>
CamelHttpQuery	String	Any query parameters, such as <code>foo=bar&beer=yes</code>
CamelHttpRawQuery	String	Camel 2.13.0: Any query parameters, such as <code>foo=bar&beer=yes</code> . Stored in the raw form, as they arrived to the consumer (i.e. before URL decoding).
CamelHttpPath	String	Additional context-path. This value is empty if the client called the context-path <code>/myapp</code> . If the client calls <code>/myapp/</code>

Name	Type	Description
		mystuff, then this header value is /mystuff. In other words its the value after the context-path configured on the route endpoint.
CamelHttpCharacterEncoding	String	The charset from the content-type header.
CamelHttpAuthentication	String	If the user was authenticated using HTTP Basic then this header is added with the value Basic.
Content-Type	String	The content type if provided. For example: text/plain; charset="UTF-8".

3.55.4. Access to Netty types

This component uses the `org.apache.camel.component.netty.http.NettyHttpRequest` as the message implementation on the [Exchange](#). This allows end users to get access to the original Netty request/response instances if needed, as shown below:

```
org.jboss.netty.handler.codec.http.HttpRequest request =
    exchange.getIn(NettyHttpRequest.class).getHttpRequest();
```

3.55.5. Examples

In the route below we use [Netty HTTP](#) as a HTTP server, which returns back a hardcoded "Bye World" message.

```
from("netty-http:http://0.0.0.0:8080/foo")
    .transform().constant("Bye World");
```

And we can call this HTTP server using Camel also, with the [ProducerTemplate](#) as shown below:

```
String out = template.requestBody("netty-http:http://localhost:8080/foo", "Hello
World", String.class);
System.out.println(out);
```

And we get back "Bye World" as the output.

3.55.6. How do I let Netty match wildcards

By default [Netty HTTP](#) will only match on exact uri's. But you can instruct Netty to match prefixes. For example

```
from("netty-http:http://0.0.0.0:8123/foo").to("mock:foo");
```

In the route above [Netty HTTP](#) will only match if the uri is an exact match, so it will match if you enter

`http://0.0.0.0:8123/foo` but not match if you do `http://0.0.0.0:8123/foo/bar`.

So if you want to enable wildcard matching you do as follows:

```
from("netty-http:http://0.0.0.0:8123/foo?matchOnUriPrefix=true").to("mock:foo");
```

So now Netty matches any endpoints with starts with `foo`.

To match **any** endpoint you can do:

```
from("netty-http:http://0.0.0.0:8123?matchOnUriPrefix=true").to("mock:foo");
```

3.55.7. Using multiple routes with same port

In the same [CamelContext](#) you can have multiple routes from [Netty HTTP](#) that shares the same port (eg a `org.jboss.netty.bootstrap.ServerBootstrap` instance). Doing this requires a number of bootstrap options to be identical in the routes, as the routes will share the same `org.jboss.netty.bootstrap.ServerBootstrap` instance. The instance will be configured with the options from the first route created.

The options the routes must be identical configured is all the options defined in the `org.apache.camel.component.netty.NettyServerBootstrapConfiguration` configuration class. If you have configured another route with different options, Camel will throw an exception on startup, indicating the options is not identical. To mitigate this ensure all options is identical.

Here is an example with two routes that share the same port.

Two routes sharing the same port:

```
from("netty-http:http://0.0.0.0:{{port}}/foo")
    .to("mock:foo")
    .transform().constant("Bye World");

from("netty-http:http://0.0.0.0:{{port}}/bar")
    .to("mock:bar")
    .transform().constant("Bye Camel");
```

And here is an example of a mis configured 2nd route that do not have identical `org.apache.camel.component.netty.NettyServerBootstrapConfiguration` option as the 1st route. This will cause Camel to fail on startup.

Two routes sharing the same port, but the 2nd route is misconfigured and will fail on starting:

```
from("netty-http:http://0.0.0.0:{{port}}/foo")
    .to("mock:foo")
    .transform().constant("Bye World");

// we cannot have a 2nd route on same port with SSL enabled, when the 1st route
is NOT
from("netty-http:http://0.0.0.0:{{port}}/bar?ssl=true")
    .to("mock:bar")
    .transform().constant("Bye Camel");
```

3.55.7.1. Reusing same server bootstrap configuration with multiple routes

By configuring the common server bootstrap option in an single instance of a `org.apache.camel.component.netty.NettyServerBootstrapConfiguration` type, we can use the `bootstrapConfiguration` option on the [Netty HTTP](#) consumers to refer and reuse the same options across all consumers.

```
<bean id="nettyHttpBootstrapOptions"
class="org.apache.camel.component.netty.NettyServerBootstrapConfiguration">
  <property name="backlog" value="200"/>
  <property name="connectionTimeout" value="20000"/>
  <property name="workerCount" value="16"/>
</bean>
```

And in the routes you refer to this option as shown below

```
<route>
  <from
```

```

uri="netty-http:http://0.0.0.0:{{port}}/foo?bootstrapConfiguration=#nettyHttpBoot
strapOptions"/>
...
</route>

<route>
  <from
    uri="netty-http:http://0.0.0.0:{{port}}/bar?bootstrapConfiguration=#nettyHttpBoot
strapOptions"/>
    ...
  </route>

<route>
  <from
    uri="netty-http:http://0.0.0.0:{{port}}/beer?bootstrapConfiguration=#nettyHttpBoot
strapOptions"/>
    ...
  </route>

```

3.55.7.2. Reusing same server bootstrap configuration with multiple routes across multiple bundles in OSGi container

See the [Netty HTTP Server Example](#) for more details and example how to do that.

3.55.8. Using HTTP Basic Authentication

The [Netty HTTP](#) consumer supports HTTP basic authentication by specifying the security realm name to use, as shown below

```

<route>
  <from
    uri="netty-http:http://0.0.0.0:{{port}}/foo?securityConfiguration.realm=karaf"/>
    ...
  </route>

```

The realm name is mandatory to enable basic authentication. By default the JAAS based authenticator is used, which will use the realm name specified (karaf in the example above) and use the JAAS realm and the JAAS `{LoginModule}`s of this realm for authentication.

End user of Apache Karaf / ServiceMix has a karaf realm out of the box, and hence why the example above would work out of the box in these containers.

3.55.8.1. Specifying ACL on web resources

The `org.apache.camel.component.netty.http.SecurityConstraint` allows to define constraints on web resources. And the `org.apache.camel.component.netty.http.SecurityConstraintMapping` is provided out of the box, allowing to easily define inclusions and exclusions with roles.

For example as shown below in the XML DSL, we define the constraint bean:

```

<bean id="constraint"
class="org.apache.camel.component.netty.http.SecurityConstraintMapping">
  <!-- inclusions defines url -> roles restrictions -->
  <!-- a * should be used for any role accepted (or even no roles) -->

```

```
<property name="inclusions">
  <map>
    <entry key="/*" value="*" />
    <entry key="/admin/*" value="admin" />
    <entry key="/guest/*" value="admin,guest" />
  </map>
</property>
<!-- exclusions is used to define public urls, which requires no authentication -->
<property name="exclusions">
  <set>
    <value>/public/*</value>
  </set>
</property>
</bean>
```

The constraint above is define so that

- access to /* is restricted and any roles is accepted (also if user has no roles)
- access to /admin/* requires the admin role
- access to /guest/* requires the admin or guest role
- access to /public/* is an exclusion which means no authentication is needed, and is therefore public for everyone without logging in

To use this constraint we just need to refer to the bean id as shown below:

```
<route>
  <from
    uri="netty-http:http://0.0.0.0:{port}/foo?matchOnUriPrefix=true&securityConfiguration.realm=karaf&securityConfiguration.securityConstraint=#constraint" />
    ...
</route>
```

3.56. OptaPlanner

Available as of Camel 2.13

The **optaplanner** component solves the planning problem contained in a message with [OptaPlanner](#). For example: feed it an unsolved Vehicle Routing problem and it solves 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-optaplanner</artifactId>
  <version>x.x.x</version><!-- use the same version as your Camel core version -->
</dependency>
```

3.56.1. URI format

```
optaplanner:solverConfig[?options]
```

The **solverConfig** is the classpath-local URI of the solverConfig, for example `/org/foo/barSolverConfig.xml`.

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.56.2. Options

No options supported yet.

3.56.3. Message Headers

No message headers supported yet.

3.56.4. Message Body

Camel takes the planning problem for the IN body, solves it and returns it on the OUT body. The IN body object must implement the `optaplanner` interface `Solution`.

3.56.5. Termination

The solving will take as long as specified in the `solverConfig`.

```
<solver>
...
<termination>
  <!-- Terminate after 10 seconds, unless it's not feasible by then yet -->
  <terminationCompositionStyle>AND</terminationCompositionStyle>
  <secondsSpentLimit>10</secondsSpentLimit>
  <bestScoreLimit>-1hard/0soft</bestScoreLimit>
</termination>
...
</solver>
```

NOTE While the Solver is solving, it will effectively hog the camel thread. Future improvements might include solving in a separate thread:

- Asynchronous solving: 1 request starts the solving and registers a callback. When the Solver terminates, the best Solution is returned through the callback.
- 2 phase request solving: 1 request starts the solving on a separate thread. Another request (with an id to the first request) terminates the Solver with `Solver.terminateEarly()` and returns the best Solution.

3.56.5.1. Samples

Solve an planning problem that's on the ActiveMQ queue with OptaPlanner:

```
from( "activemq:My.Queue" ).
  .to( "optaplanner:/org/foo/barSolverConfig.xml" );
```

Expose OptaPlanner as a REST service:

```
from( "cxfrs:bean:rsServer?bindingStyle=SimpleConsumer" )  
    .to( "optaplanner:/org/foo/barSolverConfig.xml" );
```

3.57. Properties

3.57.1. Properties Component

3.57.1.1. URI format

```
properties:key[?options]
```

where **key** is the key for the property to lookup

3.57.1.2. 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.

3.57.2. 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 [Properties](#) component

3.57.2.1. 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.

3.57.2.2. 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`: to use a specific OSGi blueprint placeholder service

3.57.2.3. 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

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
```

3.57.2.4. 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);
```

3.57.2.5. 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>
```

3.57.2.6. 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 *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](#).

3.57.2.7. 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");
```

3.57.2.8. 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");
```

3.57.2.9. Example with Simple language

The [Simple](#) language now also support using property placeholders, for example in the route below:

```
// properties
cheesy.quote=Camel rocks
```

```
// route
from("direct:start")
  .transform().simple(
    "Hi ${body} do you think ${properties:cheesy.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}.");
```

3.57.2.10. 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 property 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>
```

3.57.2.11. 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();
```

3.57.2.12. Using property placeholders for any kind of attribute in the XML DSL

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 possible 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 [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
```

3.57.2.13. Using property placeholder in the Java DSL

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");

```

3.57.2.14. Using Blueprint property placeholder with Camel routes

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 we'll -->
  <!-- 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.



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 we'll -->
    <!-- look up 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.

3.58. Quartz

The **quartz:** component provides a scheduled delivery of messages using the [Quartz Scheduler 1.x](#). Each endpoint represents a different timer (in Quartz terms, a Trigger and JobDetail).

If you are using Quartz 2.x then from Camel 2.12 onwards there is a [Quartz2](#) component you should use.

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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.58.1. URI format

```
quartz://timerName?options
quartz://groupName/timerName?options
quartz://groupName/timerName?cron=expression
quartz://timerName?cron=expression
```

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&...`

3.58.2. Options

Parameter	Default	Description
cron	<i>None</i>	Specifies a cron expression (not compatible with the <code>trigger.*</code> or <code>job.*</code> 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.
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 <code>StatefulJob</code> instead of the default job.
fireNow	false	If it is true will fire the trigger when the route is start when using SimpleTrigger.
deleteJob	true	Camel 2.12: If set to true, then the trigger automatically delete when route stop. Else if set to false, it will remain in scheduler.

Parameter	Default	Description
		When set to false, it will also mean user may reuse pre-configured trigger with camel Uri. Just ensure the names match. Notice you cannot have both deleteJob and pauseJob set to true.
pauseJob	false	Camel 2.12: If set to true, then the trigger automatically pauses when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse pre-configured trigger with camel Uri. Just ensure the names match. Notice you cannot have both deleteJob and pauseJob set to true.

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. For more information about the `StatefulJob` interface and the `JobDataMap` class, see the Quartz API documentation on <http://quartz-scheduler.org/>.

If you run in OSGi such as within Apache Karaf and have multiple bundles with Camel routes that start from Quartz endpoints, then make sure if you assign an id to the `<camelContext>` that this id is unique, as this is required by the QuartzScheduler in the OSGi container. If you do not set any id on `<camelContext>` then a unique id will be auto assigned instead.

3.58.3. Configuring quartz.properties file

By default Quartz will look for a `quartz.properties` file in the `org/quartz` directory of the classpath. If you are using WAR deployments this means just drop the `quartz.properties` in `WEB-INF/classes/org/quartz`.

However the Camel *Quartz* component also allows you to configure properties:

Parameter	Default	Type	Description
properties	null	Properties	You can configure a <code>java.util.Properties</code> 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>
```

3.58.4. Enabling Quartz scheduler in JMX

You need to configure the quartz scheduler properties to enable JMX.

That is typically setting the option `"org.quartz.scheduler.jmx.export"` to a true value in the configuration file.

From Camel 2.13 onwards Camel will automatic set this option to true, unless explicit disabled.

3.58.5. Starting the Quartz scheduler

The [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.
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>
```

3.58.6. Clustering

If you use Quartz in clustered mode, for example, the `JobStore` is clustered. Then the [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.

3.58.7. Message Headers

Camel adds the getters from the Quartz Execution Context as header values. The following headers are added: `calendar`, `fireTime`, `jobDetail`, `jobInstance`, `jobRunTime`, `mergedJobDataMap`, `nextFireTime`, `previousFireTime`, `refireCount`, `result`, `scheduledFireTime`, `scheduler`, `trigger`, `triggerName`, `triggerGroup`.

The `fireTime` header contains the `java.util.Date` of when the exchange was fired.

3.58.8. Using Cron Triggers

Quartz supports Cron-like expressions for specifying timers in a handy format. For more information, see the `CronTrigger` interface in the Quartz API documentation on <http://quartz-scheduler.org/>. 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
+	<i>Space</i>

3.59. Quartz2

Available as of Camel 2.12.0

The **quartz2:** component provides a scheduled delivery of messages using the [Quartz Scheduler 2.x](#).

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-quartz2</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

NOTE: Quartz 2.x API is not compatible with Quartz 1.x. If you need to remain on old Quartz 1.x, please use the old [Quartz](#) component instead.

3.59.1. URI format

```
quartz2://timerName?options
quartz2://groupName/timerName?options
quartz2://groupName/timerName?cron=expression
quartz2://timerName?cron=expression
```

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&...

3.59.2. Options

Parameter	Default	Description
cron	<i>None</i>	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	1000	SimpleTrigger: The amount of time in milliseconds between repeated triggers. Must enable trigger.repeatCount to use the simple trigger using this interval.
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 @PersistJobDataAfterExecution and @DisallowConcurrentExecution instead of the default job.

Parameter	Default	Description
fireNow	false	If it is true will fire the trigger when the route is start when using SimpleTrigger.
deleteJob	true	If set to true, then the trigger automatically delete when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse pre-configured trigger with camel Uri. Just ensure the names match. Notice you cannot have both deleteJob and pauseJob set to true.
pauseJob	false	If set to true, then the trigger automatically pauses when route stop. Else if set to false, it will remain in scheduler. When set to false, it will also mean user may reuse pre-configured trigger with camel Uri. Just ensure the names match. Notice you cannot have both deleteJob and pauseJob set to true.
durableJob	false	Camel 2.12.4/2.13: Whether or not the job should remain stored after it is orphaned (no triggers point to it).
recoverableJob	false	Camel 2.12.4/2.13: Instructs the scheduler whether or not the job should be re-executed if a 'recovery' or 'fail-over' situation is encountered.

For example, the following routing rule will fire two timer events to the `mock:results` endpoint:

```
from( "quartz2://myGroup/myTimerName?trigger.repeatInterval=2&trigger.repeatCount=1"
).routeId( "myRoute" ).to( "mock:result" );
```

When using `stateful=true`, the [JobDataMap](#) is re-persisted after every execution of the job, thus preserving state for the next execution.

Running in OSGi and having multiple bundles with quartz routes

If you run in OSGi such as Apache ServiceMix, or Apache Karaf, and have multiple bundles with Camel routes that start from [Quartz2](#) endpoints, then make sure if you assign an id to the `<camelContext>` that this id is unique, as this is required by the `QuartzScheduler` in the OSGi container. If you do not set any id on `<camelContext>` then a unique id is auto assigned, and there is no problem.

3.59.3. Configuring quartz.properties file

By default Quartz will look for a `quartz.properties` file in the `org/quartz` directory of the classpath. If you are using WAR deployments this means just drop the `quartz.properties` in `WEB-INF/classes/org/quartz`.

However the Camel [Quartz2](#) component also allows you to configure properties:

Parameter	Default	Type	Description
properties	null	Properties	You can configure a <code>java.util.Properties</code> 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.quartz2.QuartzComponent">
  <property name="propertiesFile" value="com/mycompany/myquartz.properties"/>
</bean>
```

3.59.4. Enabling Quartz scheduler in JMX

You need to configure the quartz scheduler properties to enable JMX.

That is typically setting the option `"org.quartz.scheduler.jmx.export"` to a true value in the configuration file.

From Camel 2.13 onwards Camel will automatic set this option to true, unless explicit disabled.

3.59.5. Starting the Quartz scheduler

The [Quartz2](#) component offers an option to let the Quartz scheduler be started delayed, or not auto started at all.

Parameter	Default	Type	Description
<code>startDelayedSeconds</code>	0	int	Seconds to wait before starting the quartz scheduler.
<code>autoStartScheduler</code>	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="quartz2" class="org.apache.camel.component.quartz2.QuartzComponent">
  <property name="startDelayedSeconds" value="5"/>
</bean>
```

3.59.6. Clustering

If you use Quartz in clustered mode, e.g. the `JobStore` is clustered. Then the [Quartz2](#) 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.

3.59.7. Message Headers

Camel adds the getters from the Quartz Execution Context as header values. The following headers are added:

`calendar`, `fireTime`, `jobDetail`, `jobInstance`, `jobRunTime`, `mergedJobDataMap`, `nextFireTime`, `previousFireTime`, `refireCount`, `result`, `scheduledFireTime`, `scheduler`, `trigger`, `triggerName`, `triggerGroup`.

The `fireTime` header contains the `java.util.Date` of when the exchange was fired.

3.59.8. 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.

For example, the following will fire a message every five minutes starting at 12pm (noon) to 6pm on weekdays:

```
from( "quartz2://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
+	<i>Space</i>

3.59.9. Specifying time zone

The Quartz Scheduler allows you to configure time zone per trigger. For example to use a timezone of your country, then you can do as follows:

```
quartz2://groupName/timerName?cron=0+0/5+12-18+?+*+MON-FRI&trigger.timeZone=Europe/Stockholm
```

The `timeZone` value is the values accepted by `java.util.TimeZone`.

3.59.10. Using QuartzScheduledPollConsumerScheduler

The [Quartz2](#) component provides a [Polling Consumer](#) scheduler which allows to use cron based scheduling for [Polling Consumer](#) such as the [File](#) and [FTP](#) consumers.

For example to use a cron based expression to poll for files every 2nd second, then a Camel route can be define simply as:

```
from("file:inbox?scheduler=quartz2&scheduler.cron=0/2+*+*+*+*+*")
    .to("bean:process");
```

Notice we define the `scheduler=quartz2` to instruct Camel to use the [Quartz2](#) based scheduler. Then we use `scheduler.xxx` options to configure the scheduler. The [Quartz2](#) scheduler requires the `cron` option to be set.

The following options is supported:

Parameter	Default	Type	Description
<code>quartzScheduler</code>	<code>null</code>	<code>org.quartz.Scheduler</code>	To use a custom Quartz scheduler. If none configure then the shared scheduler from the Quartz2 component is used.
<code>cron</code>	<code>null</code>	<code>String</code>	Mandatory: To define the cron expression for triggering the polls.
<code>triggerId</code>	<code>null</code>	<code>String</code>	To specify the trigger id. If none provided then an UUID is generated and used.
<code>triggerGroup</code>	<code>QuartzScheduledPoll-ConsumerScheduler</code>	<code>String</code>	To specify the trigger group.

Parameter	Default	Type	Description
timeZone	Default	TimeZone	The time zone to use for the CRON trigger.

Important: Remember configuring these options from the endpoint [URIs](#) must be prefixed with `scheduler..`

For example to configure the trigger id and group:

```
from("file:inbox?scheduler=quartz2&scheduler.cron=0/2+*+*+*+*?&scheduler.triggerId=myId&scheduler.triggerGroup=myGroup")
    .to("bean:process");
```

There is also a CRON scheduler in [Spring](#), so you can use the following as well:

```
from("file:inbox?scheduler=spring&scheduler.cron=0/2+*+*+*+*?")
    .to("bean:process");
```

3.60. RabbitMQ Component

Available as of Camel 2.12

The **rabbitmq:** component allows you produce and consume messages from [RabbitMQ](#) instances. Using the RabbitMQ AMQP client, this component offers a pure RabbitMQ approach over the generic [AMQP](#) 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-rabbitmq</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.60.1. URI format

```
rabbitmq://hostname[:port]/exchangeName?[options]
```

Where **hostname** is the hostname of the running rabbitmq instance or cluster. Port is optional and if not specified then defaults to the RabbitMQ client default (5672). The exchange name determines which exchange produced messages will sent to. In the case of consumers, the exchange name determines which exchange the queue will bind to.

3.60.2. Options

Property	Default	Description
autoAck	true	If messages should be auto acknowledged
autoDelete	true	If it is true, the exchange will be deleted when it is no longer in use
durable	true	If we are declaring a durable exchange (the exchange will survive a server restart)
queue	random uuid	The queue to receive messages from

Property	Default	Description
routingKey	null	The routing key to use when binding a consumer queue to the exchange. For producer routing keys, you set the header (see header section)
threadPoolSize	10	The consumer uses a Thread Pool Executor with a fixed number of threads. This setting allows you to set that number of threads.
username	null	username in case of authenticated access
password	null	password for authenticated access
vhost	/	the vhost for the channel
exchangeType	direct	Camel 2.12.2: The exchange type such as direct or topic.
bridgeEndpoint	false	Camel 2.12.3: If the bridgeEndpoint is true, the producer will ignore the message header of "rabbitmq.EXCHANGE_NAME" and "rabbitmq.ROUTING_KEY"
addresses	null	Camel 2.12.3: If this option is set, camel-rabbitmq will try to create connection based on the setting of option addresses. The addresses value is a string which looks like "server1:12345, server2:12345"
connectionTimeout	0	Camel 2.14: Connection timeout
requestedChannelMax	0	Camel 2.14: Connection requested channel max (max number of channels offered)
requestedFrameMax	0	Camel 2.14: Connection requested frame max (max size of frame offered)
requestedHeartbeat	0	Camel 2.14: Connection requested heartbeat (heart-beat in seconds offered)
sslProtocol	null	Camel 2.14: Enables SSL on connection, accepted value are `true`, `TLS` and `SSLv3`
trustManager	null	Camel 2.14: Configure SSL trust manager, SSL should be enabled for this option to be effective
clientProperties	null	Camel 2.14: Connection client properties (client info used in negotiating with the server)
connectionFactory	null	Camel 2.14: Custom RabbitMQ connection factory. When this option is set, all connection options (connectionTimeout, requestedChannelMax...) set on URI are not used
automaticRecoveryEnabled	false	Camel 2.14: Enables connection automatic recovery (uses connection implementation that performs automatic recovery when connection shutdown is not initiated by the application)
networkRecoveryInterval	5000	Camel 2.14: Network recovery interval in milliseconds (interval used when recovering from network failure)
topologyRecoveryEnabled	true	Camel 2.14: Enables connection topology recovery (should topology recovery be performed?)
prefetchEnabled	false	Camel 2.14: Enables the quality of service on the RabbitMQConsumer side, you need to specify the option of prefetchSize , prefetchCount , prefetchGlobal at the same time
prefetchSize	0	Camel 2.14: The maximum amount of content (measured in octets) that the server will deliver, 0 if unlimited.
prefetchCount	0	Camel 2.14: The maximum number of messages that the server will deliver, 0 if unlimited.
prefetchGlobal	false	Camel 2.14: If the settings should be applied to the entire channel rather than each consumer

See <http://www.rabbitmq.com/releases/rabbitmq-java-client/current-javadoc/com/rabbitmq/client/ConnectionFactory.html> and the AMQP specification for more information on connection options.

3.60.3. Custom connection factory

```
<bean id="customConnectionFactory" class="com.rabbitmq.client.ConnectionFactory">
```



```

<property name="host" value="localhost" />
<property name="port" value="5672" />
<property name="username" value="camel" />
<property name="password" value="bugsbunny" />
</bean>
<camelContext>
<route>
<from uri="direct:rabbitMQEx2" />
<to uri="rabbitmq://localhost:5672/ex2?connectionFactory=#customConnectionFactory" />
</route>
</camelContext>

```

3.60.4. Headers

The following headers are set on exchanges when consuming messages.

Property	Value
rabbitmq.ROUTING_KEY	The routing key that was used to receive the message, or the routing key that will be used when producing a message
rabbitmq.EXCHANGE_NAME	The exchange the message was received from
rabbitmq.DELIVERY_TAG	The rabbitmq delivery tag of the received message

The following headers are used by the producer. If these are set on the camel exchange then they will be set on the RabbitMQ message.

Property	Value
rabbitmq.ROUTING_KEY	The routing key that will be used when sending the message
rabbitmq.EXCHANGE_NAME	The exchange the message was received from, or sent to
rabbitmq.CONTENT_TYPE	The contentType to set on the RabbitMQ message
rabbitmq.PRIORITY	The priority header to set on the RabbitMQ message
rabbitmq.CORRELATIONID	The correlationId to set on the RabbitMQ message
rabbitmq.MESSAGE_ID	The message id to set on the RabbitMQ message
rabbitmq.DELIVERY_MODE	If the message should be persistent or not
rabbitmq.USERID	The userId to set on the RabbitMQ message
rabbitmq.CLUSTERID	The clusterId to set on the RabbitMQ message
rabbitmq.REPLY_TO	The replyTo to set on the RabbitMQ message
rabbitmq.CONTENT_ENCODING	The contentEncoding to set on the RabbitMQ message
rabbitmq.TYPE	The type to set on the RabbitMQ message
rabbitmq.EXPIRATION	The expiration to set on the RabbitMQ message
rabbitmq.TIMESTAMP	The timestamp to set on the RabbitMQ message
rabbitmq.APP_ID	The appId to set on the RabbitMQ message

Headers are set by the consumer once the message is received. The producer will also set the headers for downstream processors once the exchange has taken place. Any headers set prior to production that the producer sets will be overridden.

3.60.5. Message Body

The component will use the camel exchange in body as the rabbit mq message body. The camel exchange in object must be convertible to a byte array. Otherwise the producer will throw an exception of unsupported body type.

3.60.6. Samples

To receive messages from a queue that is bound to an exchange A with the routing key B,

```
from("rabbitmq://localhost/A?routingKey=B")
```

To receive messages from a queue with a single thread with auto acknowledge disabled.

```
from("rabbitmq://localhost/A?routingKey=B&threadPoolSize=1&autoAck=false")
```

To send messages to an exchange called C

```
...to("rabbitmq://localhost/B")
```

3.61. Ref

The **ref:** component is used for lookup of existing endpoints bound in the [Registry](#).

3.61.1. URI format

```
ref:someName[?options]
```

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.

3.61.2. 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:

```
// look up 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>
```

```
</camelContext>
```

3.61.3. 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.

3.62. RMI

The **rmi** component binds [Exchanges](#) 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 [Exchanges](#) 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.

For more information about the `Remote` interface, see the RMI API documentation, and for more information about the `Serializable` interface, see the IO API documentation on <http://docs.oracle.com/>.

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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.62.1. URI format

```
rmi://rmi-registry-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&...`

3.62.2. Options

Name	Default Value	Description
method	null	You can set the name of the method to invoke.
remoteInterfaces	null	List of interface names separated by comma.

3.62.3. Usage

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:

```
<camel:route>
  <from uri="rmi://localhost:37541/helloServiceBean?remoteInterfaces=
org.apache.camel.example.osgi.HelloService"/>
  <to uri="bean:helloServiceBean"/>
</camel:route>
```

3.63. 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>
```

Note: The component currently only supports polling (consuming) feeds.

Camel-rss internally uses a [patched version](#) of [ROME](#) hosted on ServiceMix to solve some OSGi [class loading issues](#).

3.63.1. 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&...`

3.63.2. Options

Property	Default	Description
<code>splitEntries</code>	<code>true</code>	If <code>true</code> , 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 so on. When no more entries are left in the feed, Camel contacts the remote RSS URI to obtain a new feed. If <code>false</code> , Camel obtains a fresh feed on every poll and returns all of the feed's entries.
<code>filter</code>	<code>true</code>	Use in combination with the <code>splitEntries</code> option in order to filter returned entries. By default, Camel applies the <code>UpdateDateFilter</code> 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.
<code>throttleEntries</code>	<code>true</code>	Sets whether all entries identified in a single feed poll should be delivered immediately. If <code>true</code> , only one entry is processed per <code>consumer.delay</code> . Only applicable when <code>splitEntries</code> is set to <code>true</code> .
<code>lastUpdate</code>	<code>null</code>	Use in combination with the <code>filter</code> option to block entries earlier than a specific date/time (uses the <code>entry.updated</code> timestamp). The format is: <code>yyyy-MM-ddTHH:MM:ss</code> . Example: <code>2007-12-24T17:45:59</code> .
<code>feedHeader</code>	<code>true</code>	Specifies whether to add the ROME <code>SyndFeed</code> object as a header.
<code>sortEntries</code>	<code>false</code>	If <code>splitEntries</code> is <code>true</code> , this specifies whether to sort the entries by updated date.
<code>consumer.delay</code>	<code>60000</code>	Delay in milliseconds between each poll.
<code>consumer.initialDelay</code>	<code>1000</code>	Milliseconds before polling starts.
<code>consumer.userFixedDelay</code>	<code>false</code>	Set to <code>true</code> to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.

3.63.3. 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 `SyndEntry`s.

Option	Value	Behavior
<code>splitEntries</code>	<code>true</code>	A single entry from the current feed is set in the exchange.

Option	Value	Behavior
splitEntries	false	The entire list of entries from the current feed is set in the exchange.

3.63.4. Message Headers

Header	Description
CamelRssFeed	The entire <code>SyncFeed</code> object.

3.64. Salesforce

Available as of Camel 2.12

This component supports producer and consumer endpoints to communicate with Salesforce using Java DTOs.

There is a companion maven plugin Camel Salesforce Plugin that generates these DTOs (see further below).

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-salesforce</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.64.1. URI format

The URI scheme for a salesforce component is as follows

```
salesforce:topic?options
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.64.2. Supported Salesforce APIs

The component supports the following Salesforce APIs

Producer endpoints can use the following APIs. Most of the APIs process one record at a time, the Query API can retrieve multiple Records.

3.64.2.1. Rest API

- `getVersions` - Gets supported Salesforce REST API versions

- `getResources` - Gets available Salesforce REST Resource endpoints
- `getGlobalObjects` - Gets metadata for all available SObject types
- `getBasicInfo` - Gets basic metadata for a specific SObject type
- `getDescription` - Gets comprehensive metadata for a specific SObject type
- `getObject` - Gets an SObject using its Salesforce Id
- `createObject` - Creates an SObject
- `updateObject` - Updates an SObject using Id
- `deleteObject` - Deletes an SObject using Id
- `getObjectWithId` - Gets an SObject using an external (user defined) id field
- `upsertObject` - Updates or inserts an SObject using an external id
- `deleteObjectWithId` - Deletes an SObject using an external id
- `query` - Runs a Salesforce SOQL query
- `queryMore` - Retrieves more results (in case of large number of results) using result link returned from the 'query' API
- `search` - Runs a Salesforce SOSL query

For example, the following producer endpoint uses the `upsertObject` API, with the `sObjectIdName` parameter specifying 'Name' as the external id field. The request message body should be an SObject DTO generated using the maven plugin.

The response message will either be `null` if an existing record was updated, or `CreateObjectResult` with an id of the new record, or a list of errors while creating the new object.

```
...to("salesforce:upsertObject?sObjectIdName=Name")...
```

3.64.2.2. Rest Bulk API

Producer endpoints can use the following APIs. All Job data formats, i.e. xml, csv, zip/xml, and zip/csv are supported. The request and response have to be marshalled/unmarshalled by the route. Usually the request will be some stream source like a CSV file, and the response may also be saved to a file to be correlated with the request.

- `createJob` - Creates a Salesforce Bulk Job
- `getJob` - Gets a Job using its Salesforce Id
- `closeJob` - Closes a Job
- `abortJob` - Aborts a Job
- `createBatch` - Submits a Batch within a Bulk Job
- `getBatch` - Gets a Batch using Id
- `getAllBatches` - Gets all Batches for a Bulk Job Id

- `getRequest` - Gets Request data (XML/CSV) for a Batch
- `getResults` - Gets the results of the Batch when its complete
- `createBatchQuery` - Creates a Batch from an SOQL query
- `getQueryResultIds` - Gets a list of Result Ids for a Batch Query
- `getQueryResult` - Gets results for a Result Id

For example, the following producer endpoint uses the `createBatch` API to create a Job Batch.

The in message must contain a body that can be converted into an `InputStream` (usually UTF-8 CSV or XML content from a file, etc.) and header fields `'jobId'` for the Job and `'contentType'` for the Job content type, which can be XML, CSV, ZIP_XML or ZIP_CSV. The put message body will contain `BatchInfo` on success, or throw a `SalesforceException` on error.

```
...to("salesforce:createBatchJob")..
```

3.64.2.3. Rest Streaming API

Consumer endpoints can use the following syntax for streaming endpoints to receive Salesforce notifications on create/update.

To create and subscribe to a topic

```
from("salesforce:CamelTestTopic?
notifyForFields=ALL&notifyForOperations=ALL&sObjectName=Merchandise__
c&updateTopic=true&sObjectQuery=SELECT Id, Name FROM Merchandise__c")...
```

To subscribe to an existing topic

```
from("salesforce:CamelTestTopic&sObjectName=Merchandise__c")...
```

3.64.3. Camel Salesforce Maven Plugin

This Maven plugin generates DTOs for the Camel [Salesforce](#).

3.64.3.1. Usage

The plugin configuration has the following properties.

Option	Description
<code>clientId</code>	Salesforce client Id for Remote API access
<code>clientSecret</code>	Salesforce client secret for Remote API access
<code>userName</code>	Salesforce account user name
<code>password</code>	Salesforce account password (including secret token)
<code>version</code>	Salesforce Rest API version, defaults to 25.0

Option	Description
outputDirectory	Directory where to place generated DTOs, defaults to \${project.build.directory}/generated-sources/camel-salesforce
includes	List of SObject types to include
excludes	List of SObject types to exclude
includePattern	Java RegEx for SObject types to include
excludePattern	Java RegEx for SObject types to exclude
packageName	Java package name for generated DTOs, defaults to org.apache.camel.salesforce.dto.

For obvious security reasons it is recommended that the `clientId`, `clientSecret`, `userName` and `password` fields be not set in the `pom.xml`.

The plugin should be configured for the rest of the properties, and can be executed using the following command:

```
mvn camel-salesforce:generate -DclientId=<clientId> -DclientSecret=<clientsecret> -Dusername=<username> -Dpassword=<password>
```

The generated DTOs use Jackson and XStream annotations. All Salesforce field types are supported. Date and time fields are mapped to Joda DateTime, and picklist fields are mapped to generated Java Enumerations.

3.65. SAP NetWeaver

Available as of Camel 2.12

The **sap-netweaver** integrates with the [SAP NetWeaver Gateway](#) using HTTP transports.

This camel component supports only 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-sap-netweaver</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.65.1. URI format

The URI scheme for a sap netweaver gateway component is as follows

```
sap-netweaver:https://host:8080/path?username=foo&password=secret
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

3.65.2. Prerequisites

You would need to have an account to the SAP NetWeaver system to be able to leverage this component. SAP provides a [demo setup](#) where you can requires for an account.

This component uses the basic authentication scheme for logging into SAP NetWeaver.

3.65.3. Component and endpoint options

Name	Default Value	Description
username		Username for account. This is mandatory.
password		Password for account. This is mandatory.
json	true	Whether to return data in JSON format. If this option is false, then XML is returned in Atom format.
jsonAsMap	true	To transform the JSON from a String to a Map in the message body.
flattenMap	true	If the JSON Map contains only a single entry, then flatten by storing that single entry value as the message body.

3.65.4. Message Headers

The following headers can be used by the producer.

Name	Type	Description
CamelNetWeaverCommand	String	Mandatory: The command to execute in MS ADO.Net Data Service format.

3.65.5. Examples

This example is using the flight demo example from SAP, which is available online over the internet [here](#).

In the route below we request the SAP NetWeaver demo server using the following url

```
https://sapes1.sapdevcenter.com/sap/opu/odata/IWBEP/RMTSAMPLEFLIGHT_2/
```

And we want to execute the following command

```
FlightCollection(AirLineID=&#39;AA&#39;;FlightConnectionID=&#39;0017&#39;;FlightDate=datetime&#39;2012-08-29T00%3A00%3A00&#39;)
```

To get flight details for the given flight. The command syntax is in [MS ADO.Net Data Service](#) format.

We have the following Camel route

```
from("direct:start")
  .toF("sap-netweaver:%s?username=%s&password=%s", url, username, password)
  .to("log:response")
  .to("velocity:flight-info.vm")
```

Where url, username, and password is defined as:

```
private String username = "P1909969254";
private String password = "TODO";
private String url =
"https://sapes1.sapdevcenter.com/sap/opu/odata/IWBEP/RMTSAMPLEFLIGHT_2/" ;
```

```
private String command =
"FlightCollection(AirLineID=&#39;AA&#39;,FlightConnectionID=&#39;0017&#39;,FlightDate=dateTime&#39;2012-08-29T00%3A00%3A00&#39;);";
```

The password is invalid. You would need to create an account at SAP first to run the demo.

The velocity template is used for formatting the response to a basic HTML page

```
<html>
  <body>
    Flight information:

    <p/>
    <br/>Airline ID: $body["AirLineID"]
    <br/>Aircraft Type: $body["AirCraftType"]
    <br/>Departure city: $body["FlightDetails"]["DepartureCity"]
    <br/>Departure airport: $body["FlightDetails"]["DepartureAirPort"]
    <br/>Destination city: $body["FlightDetails"]["DestinationCity"]
    <br/>Destination airport: $body["FlightDetails"]["DestinationAirPort"]

  </body>
</html>
```

When running the application you get sampel output:

```
Flight information:
Airline ID: AA
Aircraft Type: 747-400
Departure city: new york
Departure airport: JFK
Destination city: SAN FRANCISCO
Destination airport: SFO
```

3.66. 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 [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 [JMS](#) or [ActiveMQ](#).



The [Direct](#) component provides synchronous invocation of any consumers when a producer sends a message exchange.

3.66.1. 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 SEDA queue name **must** be used for both the producer endpoint and the consumer endpoint. Otherwise Camel will create a second [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");
```

And where URI *option* can be:

Table 3.18.

Name	Default	Description
size		The maximum capacity of the SEDA queue (i.e., the number of messages it can hold). The default value in Camel 2.2 or older is 1000. From Camel 2.3 onwards, the size is unbounded by default. Notice: Mind if you use this option, then its the first endpoint being created with the queue name, that determines the size. To make sure all endpoints use same size, then configure the size option on all of them, or the first endpoint being created. From Camel 2.11 onwards, a validation is taken place to ensure if using mixed queue sizes for the same queue name, Camel would detect this and fail creating the endpoint.
concurrent-Consumers	1	Number of concurrent threads processing 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 Request Reply based. The default option is IfReplyExpected. See more information about Async messaging.
timeout	30000	Timeout in milliseconds a seda producer will at most waiting for an async task to complete. See <code>waitForTaskToComplete</code> and Async for more details. You can disable timeout by using 0 or a negative value.
multipleConsumers	false	Specifies whether multiple consumers are allowed. If enabled, you can use SEDA for Publish-Subscribe messaging. That is, you can send a message to the SEDA queue and have each consumer receive a copy of the message. When enabled, this option should be specified on every consumer endpoint.
limitConcurrent-Consumers	true	Whether to limit the <code>concurrentConsumers</code> to maximum 500. By default, an exception will be thrown if a SEDA endpoint is configured with a greater number. You can disable that check by turning this option off.
blockWhenFull	false	Whether a thread that sends messages to a full SEDA queue will block until the queue's capacity is no longer exhausted. By default, an exception will be thrown stating that the queue is full. By enabling this option, the calling thread will instead block and wait until the message can be accepted.
queueSize		Component only. The maximum default size (capacity of the number of messages it can hold) of the SEDA queue. This option is used if <code>size</code> is not in use.
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.
purgeWhenStopping	false	Whether to purge the task queue when stopping the consumer/route. This allows to stop faster, as any pending messages on the queue is discarded.
queue	null	Define the queue instance which will be used by seda endpoint
queueFactory	null	Define the QueueFactory which could create the queue for the seda endpoint

Name	Default	Description
failIfNoConsumers	false	Whether the producer should fail by throwing an exception, when sending to a SEDA queue with no active consumers.

See the [Camel Website](#) for the most up-to-date examples of this component in use.

3.66.1.1. Choosing BlockingQueue implementation

Available as of Camel 2.12

By default, the SEDA component always instantiates `LinkedBlockingQueue`, but you can use different implementation, you can reference your own `BlockingQueue` implementation, in this case the size option is not used

```
<bean id="arrayQueue" class="java.util.ArrayBlockingQueue">
<constructor-arg index="0" value="10" ><!-- size -->
<constructor-arg index="1" value="true" ><!-- fairness -->
</bean>
<!-- ... and later -->
<from>seda:array?queue=#arrayQueue</from>
```

Or you can reference a `BlockingQueueFactory` implementation, 3 implementations are provided `LinkedBlockingQueueFactory`, `ArrayBlockingQueueFactory` and `PriorityBlockingQueueFactory`:

```
<bean id="priorityQueueFactory"
class="org.apache.camel.component.seda.PriorityBlockingQueueFactory">
<property name="comparator">
<bean class="org.apache.camel.demo.MyExchangeComparator" />
</property>
</bean>
<!-- ... and later -->
<from>seda:priority?queueFactory=#priorityQueueFactory&size=100</from>
```

3.66.2. Use of Request Reply

The [SEDA](#) component supports using [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 [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 [Request Reply](#) over [SEDA](#) or [VM](#), you can chain as many endpoints as you like.

3.66.3. 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.

3.66.4. 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 [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.

3.67. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</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.

3.67.1. URI format and options

```
servlet://relative_path[?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`, where *option* can be:

Table 3.19.

Name	Default Value	Description
<code>httpBindingRef</code>	<code>null</code>	Reference to an Camel <code>HttpBinding</code> object in the Registry . A <code>HttpBinding</code> 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.

3.67.2. Message Headers

Camel will apply the same Message Headers as the [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.

3.67.3. 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](#).

3.67.4. 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>");
    }
});

```



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.

3.68. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>

```

3.68.1. 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:*
```

3.68.2. 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);
```

3.68.3. 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 that 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

Name	Default Value	Type	Description
			further action i.e continue further on the route. If no Permissions list or Roles List (see below) is provided to the ShiroSecurityPolicy object, then authorization is deemed as not required. Note that the default is that authorization is granted if any of the Permission Objects in the list are applicable.
rolesList	none	List<String>	Camel 2.13: A List of roles required in order for an authenticated user to be authorized to perform further action i.e continue further on the route. If no roles list or permissions list (see above) is provided to the ShiroSecurityPolicy object, then authorization is deemed as not required. Note that the default is that authorization is granted if any of the roles in the list are applicable.
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
base64	false	boolean	Camel 2.12: To use base64 encoding for the security token header, which allows transferring the header over JMS etc. This option must also be set on ShiroSecurityTokenInjector as well.
allPermissionsRequired	false	boolean	Camel 2.13: The default is that authorization is granted if any of the Permission Objects in the permissionsList parameter are applicable. Set this to true to require all of the Permissions to be met.
allRolesRequired	false	boolean	Camel 2.13: The default is that authorization is granted if any of the roles in the rolesList parameter are applicable. Set this to true to require all of the roles to be met.

3.68.4. 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");
        }
    };
}
```

```
}
```

3.68.5. 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");
        }
    };
}
```

3.68.6. 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");
```

3.68.7. 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();
}
```

3.68.8. Sending Messages to routes secured by a ShiroSecurityPolicy (much easier from Camel 2.12 onwards)

From **Camel 2.12** onwards its even easier as you can provide the subject in two different ways.

3.68.8.1. Using ShiroSecurityToken

You can send a message to a Camel route with a header of key `ShiroSecurityConstants.SHIRO_SECURITY_TOKEN` of the type `org.apache.camel.component.shiro.security.ShiroSecurityToken` that contains the username and password. For example:

```
ShiroSecurityToken shiroSecurityToken = new ShiroSecurityToken("ringo", "starr");

template.sendBodyAndHeader("direct:secureEndpoint", "Beatle Mania",
    ShiroSecurityConstants.SHIRO_SECURITY_TOKEN, shiroSecurityToken);
```

You can also provide the username and password in two different headers as shown below:

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put(ShiroSecurityConstants.SHIRO_SECURITY_USERNAME, "ringo");
headers.put(ShiroSecurityConstants.SHIRO_SECURITY_PASSWORD, "starr");
template.sendBodyAndHeaders("direct:secureEndpoint", "Beatle Mania", headers);
```

When you use the username and password headers, then the ShiroSecurityPolicy in the Camel route will automatic transform those into a single header with key `ShiroSecurityConstants.SHIRO_SECURITY_TOKEN` with the

token. Then token is either a `ShiroSecurityToken` instance, or a base64 representation as a `String` (the latter is when you have set `base64=true`).

3.69. SJMS

Available as of Camel 2.11

The Simple JMS Component, or SJMS, is a JMS client for use with Camel that uses well known best practices when it comes to JMS client creation and configuration. SJMS contains a brand new JMS client API written explicitly for Camel eliminating third party messaging implementations keeping it light and resilient. The following features is included:

- Standard Queue and Topic Support (Durable & Non-Durable)
- InOnly & InOut MEP Support
- Asynchronous Producer and Consumer Processing
- Internal JMS Transaction Support

Additional key features include:

- Pluggable Connection Resource Management
- Session, Consumer, & Producer Pooling & Caching Management
- Batch Consumers and Producers
- Transacted Batch Consumers & Producers
- Support for Customizable Transaction Commit Strategies (Local JMS Transactions only)

Why the S in SJMS

S stands for Simple and Standard and Springless. Also camel-jms was already taken.

This is a rather new component in a complex world of JMS messaging. So this component is ongoing development and hardening.

The classic [JMS](#) component based on Spring JMS has been hardened and battle tested extensively.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sjms</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.69.1. URI format

```
sjms:[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:

```
sjms:FOO.BAR
```

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

```
sjms:queue:FOO.BAR
```


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

```
sjms:topic:Stocks.Prices
```

You append query options to the URI using the following format, `?option=value&option=value&...`

3.69.2. Component Options and Configurations

The SJMS Component supports the following configuration options:

Option	Required	Default Value	Description
<code>connectionCount</code>		1	The maximum number of connections available to endpoints started under this component
<code>connectionFactory</code>		null	A ConnectionFactory is required to enable the <code>SjmsComponent</code> . It can be set directly or set as part of a <code>ConnectionFactoryResource</code> .
<code>connectionResource</code>		null	A <code>ConnectionFactoryResource</code> is an interface that allows for customization and container control of the <code>ConnectionFactory</code> . See Pluggable Connection Resource Management for further details.
<code>headerFilter-Strategy</code>		<code>DefaultJmsKey-FormatStrategy</code>	
<code>keyFormatStrategy</code>		<code>DefaultJmsKey-FormatStrategy</code>	
<code>transactionCommit-Strategy</code>		null	

Below is an example of how to configure the `SjmsComponent` with its required `ConnectionFactory` provider. It will create a single connection by default and store it using the components internal pooling APIs to ensure that it is able to service Session creation requests in a thread safe manner.

```
SjmsComponent component = new SjmsComponent();
component.setConnectionFactory(new
ActiveMQConnectionFactory("tcp://localhost:61616"));
getContext().addComponent("sjms", component);
```

For a `SjmsComponent` that is required to support a durable subscription, you can override the default `ConnectionFactoryResource` instance and set the `clientId` property.

```
ConnectionFactoryResource connectionResource = new ConnectionFactoryResource();
connectionResource.setConnectionFactory(new
ActiveMQConnectionFactory("tcp://localhost:61616"));
connectionResource.setClientId("myclient-id");
```

```
SjmsComponent component = new SjmsComponent();
component.setConnectionResource(connectionResource);
component.setMaxConnections(1);
```

3.69.3. Producer Configuration Options

The SjmsProducer Endpoint supports the following properties:

Option	Default Value	Description
acknowledgementMode	AUTO_ACKNOWLEDGE	The JMS acknowledgement name, which is one of: SESSION_TRANSACTED, AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE. CLIENT_ACKNOWLEDGE is not supported at this time.
consumerCount	1	InOut only. Defines the number of MessageListener instances that for response consumers.
exchangePattern	InOnly	Sets the Producers message exchange pattern.
namedReplyTo	null	InOut only. Specifies a named reply to destination for responses.
persistent	true	Whether a message should be delivered with persistence enabled.
producerCount	1	Defines the number of MessageProducer instances.
responseTimeout	5000	InOut only. Specifies the amount of time an InOut Producer will wait for its response.
synchronous	true	Sets whether the Endpoint will use synchronous or asynchronous processing.
transacted	false	If the endpoint should use a JMS Session transaction.
ttl	-1	Disabled by default. Sets the Message time to live header.

3.69.4. Producer Usage

3.69.4.1. InOnly Producer - (Default)

The InOnly Producer is the default behavior of the SJMS Producer Endpoint.

```
from("direct:start")
    .to("sjms:queue:bar");
```

3.69.4.2. InOut Producer

To enable InOut behavior append the `exchangePattern` attribute to the URI. By default it will use a dedicated `TemporaryQueue` for each consumer.

```
from("direct:start")
    .to("sjms:queue:bar?exchangePattern=InOut&namedReplyTo=my.reply.to.queue");
```

You can specify a `namedReplyTo` though which can provide a better monitor point.

```
from( "direct:start" )
    .to( "sjms:queue:bar?exchangePattern=InOut&namedReplyTo=my.reply.to.queue" );
```

3.69.5. Consumers Configuration Options

The `SjmsConsumer` Endpoint supports the following properties:

Option	Default Value	Description
<code>acknowledgementMode</code>	<code>AUTO_ACKNOWLEDGE</code>	The JMS acknowledgement name, which is one of: <code>TRANSACTIONAL</code> , <code>AUTO_ACKNOWLEDGE</code> or <code>DUPS_OK_ACKNOWLEDGE</code> . <code>CLIENT_ACKNOWLEDGE</code> is not supported at this time.
<code>consumerCount</code>	<code>1</code>	Defines the number of MessageListener instances.
<code>durableSubscriptionId</code>	<code>null</code>	Required for a durable subscriptions.
<code>exchangePattern</code>	<code>InOnly</code>	Sets the Consumers message exchange pattern.
<code>messageSelector</code>	<code>null</code>	Sets the message selector.
<code>synchronous</code>	<code>true</code>	Sets whether the Endpoint will use synchronous or asynchronous processing.
<code>transacted</code>	<code>false</code>	If the endpoint should use a JMS Session transaction.
<code>transactionBatchCount</code>	<code>1</code>	The number of exchanges to process before committing a local JMS transaction. The <code>transacted</code> property must also be set to <code>true</code> or this property will be ignored.
<code>transactionBatchTimeout</code>	<code>5000</code>	The amount of time a the transaction will stay open between messages before committing what has already been consumed. Minimum value is 1000ms.
<code>ttl</code>	<code>-1</code>	Disabled by default. Sets the Message time to live header.

3.69.6. Consumer Usage

3.69.6.1. InOnly Consumer - (Default)

The `InOnly` Consumer is the default Exchange behavior of the `SJMS` Consumer Endpoint.

```
from( "sjms:queue:bar" )
    .to( "mock:result" );
```

3.69.6.2. InOut Consumer

To enable `InOut` behavior append the `exchangePattern` attribute to the URI.

```
from( "sjms:queue:in.out.test?exchangePattern=InOut" )
    .transform( constant( "Bye Camel" ) );
```


3.69.7. Advanced Usage Notes

3.69.7.1. Pluggable Connection Resource Management

SJMS provides JMS [Connection](#) resource management through built-in connection pooling. This eliminates the need to depend on third party API pooling logic. However there may be times that you are required to use an external Connection resource manager such as those provided by J2EE or OSGi containers. For this SJMS provides an interface that can be used to override the internal SJMS Connection pooling capabilities. This is accomplished through the [ConnectionResource](#) interface.

The [ConnectionResource](#) provides methods for borrowing and returning Connections as needed is the contract used to provide [Connection](#) pools to the SJMS component. A user should use when it is necessary to integrate SJMS with an external connection pooling manager.

It is recommended though that for standard [ConnectionFactory](#) providers you use the [ConnectionFactoryResource](#) implementation that is provided with SJMS as-is or extend as it is optimized for this component.

Below is an example of using the pluggable ConnectionResource with the ActiveMQ PooledConnectionFactory:

```
public class AMQConnectionResource implements ConnectionResource {
    private PooledConnectionFactory pcf;

    public AMQConnectionResource(String connectString, int maxConnections) {
        super();
        pcf = new PooledConnectionFactory(connectString);
        pcf.setMaxConnections(maxConnections);
        pcf.start();
    }

    public void stop() {
        pcf.stop();
    }

    @Override
    public Connection borrowConnection() throws Exception {
        Connection answer = pcf.createConnection();
        answer.start();
        return answer;
    }

    @Override
    public Connection borrowConnection(long timeout) throws Exception {
        // SNIPPED...
    }

    @Override
    public void returnConnection(Connection connection) throws Exception {
        // Do nothing since there isn't a way to return a Connection
        // to the instance of PooledConnectionFactory
        log.info("Connection returned");
    }
}
```

Then pass in the ConnectionResource to the SjsComponent:

```
CamelContext camelContext = new DefaultCamelContext();
AMQConnectionResource pool = new AMQConnectionResource("tcp://localhost:33333", 1);
SjsComponent component = new SjsComponent();
component.setConnectionResource(pool);
camelContext.addComponent("sjms", component);
```

To see the full example of its usage please refer to the [ConnectionResourceIT](#).

3.69.7.2. Session, Consumer, & Producer Pooling & Caching Management

Coming soon ...

3.69.7.3. Batch Message Support

The `SjmsProducer` supports publishing a collection of messages by creating an `Exchange` that encapsulates a `List`. This `SjmsProducer` will then iterate through the contents of the `List` and publish each message individually.

If when producing a batch of messages there is the need to set headers that are unique to each message you can use the `SJMS BatchMessage` class. When the `SjmsProducer` encounters a `BatchMessage List` it will iterate each `BatchMessage` and publish the included payload and headers.

Below is an example of using the `BatchMessage` class. First we create a `List` of `BatchMessages`:

```
List<BatchMessage<String>> messages = new ArrayList<BatchMessage<String>>();
for (int i = 1; i <= messageCount; i++) {
    String body = "Hello World " + i;
    BatchMessage<String> message = new BatchMessage<String>(body, null);
    messages.add(message);
}
```

Then publish the `List`:

```
template.sendBody("sjms:queue:batch.queue", messages);
```

3.69.7.4. Customizable Transaction Commit Strategies (Local JMS Transactions only)

`SJMS` provides a developer the means to create a custom and pluggable transaction strategy through the use of the `TransactionCommitStrategy` interface. This allows a user to define a unique set of circumstances that the `SessionTransactionSynchronization` will use to determine when to commit the `Session`. An example of its use is the `BatchTransactionCommitStrategy` which is detailed further in the next section.

3.69.7.5. Transacted Batch Consumers & Producers

The `SjmsComponent` has been designed to support the batching of local JMS transactions on both the `Producer` and `Consumer` endpoints. How they are handled on each is very different though.

The `SjmsConsumer` endpoint is a straightforward implementation that will process `X` messages before committing them with the associated `Session`. To enable batched transaction on the consumer first enable transactions by setting the `transacted` parameter to `true` and then adding the `transactionBatchCount` and setting it to any value that is greater than 0. For example the following configuration will commit the `Session` every 10 messages:

```
sjms:queue:transacted.batch.consumer?transacted=true&transactionBatchCount=10
```

If an exception occurs during the processing of a batch on the consumer endpoint, the `Session` rollback is invoked causing the messages to be redelivered to the next available consumer. The counter is also reset to 0 for the `BatchTransactionCommitStrategy` for the associated `Session` as well. It is the responsibility of the user to ensure they put hooks in their processors of batch messages to watch for messages with the `JMSRedelivered` header set to `true`. This is the indicator that messages were rolled back at some point and that a verification of a successful processing should occur.

A transacted batch consumer also carries with it an instance of an internal timer that waits a default amount of time (5000ms) between messages before committing the open transactions on the Session. The default value of 5000ms (minimum of 1000ms) should be adequate for most use-cases but if further tuning is necessary simply set the `transactionBatchTimeout` parameter.

```
sjms:queue:transacted.batch.consumer?transacted=true&transactionBatchCount=10&transactionBatchTimeout=2000
```

The minimal value that will be accepted is 1000ms as the amount of context switching may cause unnecessary performance impacts without gaining benefit.

The producer endpoint is handled much differently though. With the producer after each message is delivered to its destination the Exchange is closed and there is no longer a reference to that message. To make all the messages available for redelivery you simply enable transactions on a Producer Endpoint that is publishing BatchMessages. The transaction will commit at the conclusion of the exchange which includes all messages in the batch list. Nothing additional need be configured. For example:

```
List<BatchMessage<String>> messages = new ArrayList<BatchMessage<String>>();
for (int i = 1; i <= messageCount; i++) {
    String body = "Hello World " + i;
    BatchMessage<String> message = new BatchMessage<String>(body, null);
    messages.add(message);
}
```

Now publish the List with transactions enabled:

```
template.sendBody("sjms:queue:batch.queue?transacted=true", messages);
```

3.69.8. Additional Notes

3.69.8.1. Message Header Format

The SJMS Component uses the same header format strategy that is used in the Camel JMS Component. This pluggable strategy ensures that messages sent over the wire 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 & hyphens and the reverse when consuming JMS messages:

- is replaced by *DOT* and the reverse replacement when Camel consumes the message.
- is replaced by *HYPHEN* and the reverse replacement when Camel consumes the message.

See also the option `jmsKeyFormatStrategy`, which allows use of your own custom strategy for formatting keys.

For the `exchange.in.header`, the following rules apply for the header values:

3.69.8.2. Message Content

To deliver content over the wire we must ensure that the body of the message that is being delivered adheres to the JMS Message Specification. Therefore, all that are produced must either 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.

3.69.8.3. Clustering

When using InOut with SJMS in a clustered environment you must either use TemporaryQueue destinations or use a unique named reply to destination per InOut producer endpoint. Message correlation is handled by the endpoint, not with message selectors at the broker. The InOut Producer Endpoint uses Java Concurrency Exchangers cached by the Message JMSCorrelationID. This provides a nice performance increase while reducing the overhead on the broker since all the messages are consumed from the destination in the order they are produced by the interested consumer.

Currently the only correlation strategy is to use the JMSCorrelationId. The InOut Consumer uses this strategy as well ensuring that all responses messages to the included JMSReplyTo destination also have the JMSCorrelationId copied from the request as well.

3.69.9. Transaction Support

SJMS currently only supports the use of internal JMS Transactions. There is no support for the Camel Transaction Processor or the Java Transaction API (JTA).

3.69.9.1. Does Springless Mean I Can't Use Spring?

Not at all. Below is an example of the SJMS component using the Spring DSL:

```
<route
  id="inout.named.reply.to.producer.route">
  <from
    uri="direct:invoke.named.reply.to.queue" />
  <to
    uri="sjms:queue:named.reply.to.queue?namedReplyTo=my.response.queue&
    exchangePattern=InOut" />
</route>
```

Springless refers to moving away from the dependency on the Spring JMS API. A new JMS client API is being developed from the ground up to power SJMS.

3.70. 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>
```

```
<!-- use the same version as your Camel core version -->
<version>x.x.x</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
```

3.70.1. URI Format and options

```
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 will 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&...`, where *option* can be:

Table 3.20.

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

Name	Default Value	Description
sourceAddr	1616	Defines the address of SME (Short Message Entity) which originated this message.
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. Starting with Camel 2.11, Camel will check the in message headers 'CamelSmppSystemId' and 'CamelSmppPassword' of the first exchange. If they are present, Camel will use this information to connect to the SMSC.
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.
sessionStateListener	null	You can refer to a org.jsmpp. session. SessionStateListener in the Registry to receive callbacks when the session state changed.
addressRange	null	Starting with Camel 2.11, you can specify the address range for the SmppConsumer as defined in section 5.2.7 of the SMPP 3.4 specification. The SmppConsumer will receive messages only from SMSC's which target an address (MSISDN or IP address) within this range.

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
```

3.70.2. 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 <code>sourceAddrTon</code> 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 <code>sourceAddrNpi</code> 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 <code>sourceAddrTon</code> URI options listed above.
CamelSmppSourceAddrNpi	Byte	Defines the numeric plan indicator (NPI) to be used in the SME originator address parameters. Same as the <code>sourceAddrNpi</code> 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 <code>serviceType</code> URI options listed above.
CamelSmppRegistered Delivery	Byte	Only for SubmitSm, SubmitMulti, CancelSm and DataSm. Same as the <code>registeredDelivery</code> URI options listed above.
CamelSmppPriorityFlag	Byte	Only for SubmitSm and SubmitMulti. Same as the <code>priorityFlag</code> 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 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 CamelSmppDataCoding	/ Byte	Only for SubmitSm, SubmitMulti and ReplaceSm. Same as the <code>alphabet</code> URI options listed above.
CamelSmppOptionalParameters	Map<String, String>	Deprecated and will be removed in Camel 2.13.0/3.0.0

Header	Type	Description
		Camel 2.10.5 and 2.11.1 onwards and only for SubmitSm, SubmitMulti and DataSm The optional parameters send back by the SMSC.
CamelSmppOptionalParameter	Map<Short, Object>	Camel 2.10.7 and 2.11.2 onwards and only for SubmitSm, SubmitMulti and DataSm The optional parameter which are send to the SMSC. The value is converted in the following way: String -> org.jsmpp.bean. OptionalParameter.COctetString byte[] -> org.jsmpp.bean. OptionalParameter.OctetString Byte -> org.jsmpp.bean. OptionalParameter.Byte Integer -> org.jsmpp.bean. OptionalParameter.Int Short -> org.jsmpp.bean. OptionalParameter.Short null -> org.jsmpp.bean. OptionalParameter.Null

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)).
CamelSmppOptionalParameters	Map<String, String>	Deprecatd and will be removed in Camel 2.13.0/3.0.0 From Camel 2.11.1 onwards only for DataSm The optional parameters which are returned from the SMSC by sending the message.
CamelSmppOptionalParameter	Map<Short, Object>	From Camel 2.10.7, 2.11.2 onwards only for DataSm The optional parameter which are returned from the SMSC by sending the message. The key is the Short code for the optional parameter. The value is converted in the following way: org.jsmpp.bean.Optional

Header	Type	Description
		Parameter.COctetString -> String org.jsmpp.bean.Optional Parameter.OctetString -> byte[] org.jsmpp.bean.Optional Parameter.Byte -> Byte org.jsmpp.bean.Optional Parameter.Int -> Integer org.jsmpp.bean.Optional Parameter.Short -> Short org.jsmpp.bean.Optional Parameter.Null -> null

3.70.3. 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.
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 <code>sourceAddrNpi</code> 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 <code>sourceAddrTon</code> 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 <code>sourceAddrNpi</code> 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 <code>sourceAddrTon</code> 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: <code>DELIVRD</code> : Message is delivered to

Header	Description
	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.
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 <code>registeredDelivery</code> URI options listed above.
CamelSmppDestAddrNpi	Only for DataSm. Defines the numeric plan indicator (NPI) in the destination address parameters. Same as the <code>sourceAddrNpi</code> URI options listed above.
CamelSmppDestAddrTon	Only for DataSm. Defines the type of number (TON) in the destination address parameters. Same as the <code>sourceAddrTon</code> URI options listed above.
CamelSmppMessageType	Identifies the type of an incoming message: <code>AlertNotification</code> : an SMSC alert notification <code>DataSm</code> : an SMSC data short message <code>DeliveryReceipt</code> : an SMSC delivery receipt <code>DeliverSm</code> : an SMSC deliver short message
CamelSmppOptionalParameters	A <code>Map<String, Object></code> . Starting with Camel 2.10.5 onwards, only for <code>DeliverSm</code> . The optional parameters sent back by the SMSC.



See the documentation of the [JSMPP Library](#) for more details about the underlying library.

3.70.4. 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&
enquireLinkTimer=3000&transactionTimer=5000&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&
enquireLinkTimer=3000&transactionTimer=5000&systemType=consumer"/>
  <to uri="bean:foo"/>
</route>
```



If you need an SMSC simulator for your test, you can use the simulator provided by [Logica](#).

3.71. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.71.1. 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&...`

3.71.2. Options

Name	Default Value	Description
type	none	The type of action you want to perform. You can enter here <code>POLL</code> or <code>TRAP</code> . The value <code>POLL</code> will instruct the endpoint to poll a given host for the supplied OID keys. If you put in <code>TRAP</code> you will setup a listener for SNMP Trap Events.
protocol	udp	Here you can select which protocol to use. You can use either <code>udp</code> or <code>tcp</code> .
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.

Name	Default Value	Description
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"

3.71.3. 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.

3.71.4. 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
```

You can get the community of SNMP TRAP with message header 'securityName', and the peer address of the SNMP TRAP with message header 'peerAddress'.

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" );
```

3.72. Solr

Available as of Camel 2.9

The Solr component allows you to interface with an [Apache Lucene Solr](#) server (based on SolrJ 3.5.0).

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-solr</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.72.1. URI format

```
solr://host[:port]/solr?[options]
```

3.72.2. Endpoint Options

The following SolrServer options may be configured on the Solr endpoint.

name	default value	description
maxRetries	0	maximum number of retries to attempt in the event of transient errors
soTimeout	1000	read timeout on the underlying <code>HttpConnectionManager</code> . This is desirable for queries, but probably not for indexing
connectionTimeout	100	connectionTimeout on the underlying <code>HttpConnectionManager</code>
defaultMaxConnectionsPerHost	2	maxConnectionsPerHost on the underlying <code>HttpConnectionManager</code>
maxTotalConnections	20	maxTotalConnection on the underlying <code>HttpConnectionManager</code>
followRedirects	false	indicates whether redirects are used to get to the Solr server
allowCompression	false	server side must support gzip or deflate for this to have any effect
requestHandler	/update (xml)	set the request handler to be used
streamingThreadCount	2	Camel 2.9.2 set the number of threads for the StreamingUpdateSolrServer
streamingQueueSize	10	Camel 2.9.2 set the queue size for the StreamingUpdateSolrServer

3.72.3. Message Operations

The following Solr operations are currently supported. Simply set an exchange header with a key of "SolrOperation" and a value set to one of the following. Some operations also require the message body to be set.

- the INSERT operations use the [CommonsHttpSolrServer](#)
- the INSERT_STREAMING operations use the [StreamingUpdateSolrServer](#) (**Camel 2.9.2**)

operation	message body	description
INSERT/ INSERT_STREAMING	n/a	adds an index using message headers (must be prefixed with "SolrField.")
INSERT/ INSERT_STREAMING	File	adds an index using the given File (using ContentStreamUpdateRequest)
INSERT/ INSERT_STREAMING	SolrInputDocument	Camel 2.9.2 updates index based on the given SolrInputDocument
INSERT/ INSERT_STREAMING	String XML	Camel 2.9.2 updates index based on the given XML (must follow SolrInputDocument format)
ADD_BEAN	bean instance	adds an index based on values in an annotated bean
DELETE_BY_ID	index id to delete	delete a record by ID
DELETE_BY_QUERY	query string	delete a record by a query
COMMIT	n/a	performs a commit on any pending index changes
ROLLBACK	n/a	performs a rollback on any pending index changes
OPTIMIZE	n/a	performs a commit on any pending index changes and then runs the optimize command

3.72.4. Example

Below is a simple INSERT, DELETE and COMMIT example

```
from("direct:insert")
    .setHeader(SolrConstants.OPERATION, constant(SolrConstants.OPERATION_INSERT))
    .setHeader(SolrConstants.FIELD + "id", body())
    .to("solr://localhost:8983/solr");

from("direct:delete")
    .setHeader(SolrConstants.OPERATION,
        constant(SolrConstants.OPERATION_DELETE_BY_ID))
    .to("solr://localhost:8983/solr");

from("direct:commit")
    .setHeader(SolrConstants.OPERATION, constant(SolrConstants.OPERATION_COMMIT))
    .to("solr://localhost:8983/solr");
```

```
<route>
  <from uri="direct:insert"/>
  <setHeader headerName="SolrOperation">
    <constant>INSERT</constant>
  </setHeader>
  <setHeader headerName="SolrField.id">
    <simple>${body}</simple>
  </setHeader>
  <to uri="solr://localhost:8983/solr"/>
</route>
<route>
  <from uri="direct:delete"/>
  <setHeader headerName="SolrOperation">
    <constant>DELETE_BY_ID</constant>
```

```

    </setHeader>
    <to uri="solr://localhost:8983/solr"/>
</route>
<route>
    <from uri="direct:commit"/>
    <setHeader headerName="SolrOperation">
        <constant>COMMIT</constant>
    </setHeader>
    <to uri="solr://localhost:8983/solr"/>
</route>

```

A client would simply need to pass a body message to the insert or delete routes and then call the commit route.

```

template.sendBody("direct:insert", "1234");
template.sendBody("direct:commit", null);
template.sendBody("direct:delete", "1234");
template.sendBody("direct:commit", null);

```

3.72.5. Querying Solr

Currently, this component doesn't support querying data natively (may be added later). For now, you can query Solr using [HTTP](#) as follows:

```

//define the route to perform a basic query
from("direct:query")
    .recipientList(simple("http://localhost:8983/solr/select/?q=${body}"))
    .convertBodyTo(String.class);
...
//query for an id of '1234' (url encoded)
String responseXml = (String) template.requestBody("direct:query", "id%3A1234");

```

For more information, see these resources...

[Solr Query Tutorial](#)

[Solr Query Syntax](#)

3.73. Splunk

Available as of Camel 2.13

The Splunk component provides access to [Splunk](#) using the Splunk provided [client](#) api, and it enables you to publish and search for events in Splunk.

Maven users will need to add the following dependency to their pom.xml for this component:

```

<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-splunk</artifactId>
    <version>${camel-version}</version>
</dependency>

```

3.73.1. URI format

```
splunk://[endpoint]?[options]
```

3.73.2. Producer Endpoints:

Endpoint	Description
stream	Streams data to a named index or the default if not specified. When using stream mode be aware of that Splunk has some internal buffer (about 1MB or so) before events gets to the index. If you need realtime, better use submit or tcp mode.
submit	submit mode. Uses Splunk rest api to publish events to a named index or the default if not specified.
tcp	tcp mode. Streams data to a tcp port, and requires a open receiver port in Splunk.

When publishing events the message body should contain a SplunkEvent.

Example

```
from("direct:start").convertBodyTo(SplunkEvent.class)
    .to("splunk://submit?username=user&password=123&index=myindex&sourceType=
someSourceType&source=mySource")...
```

In this example a converter is required to convert to a SplunkEvent class.

3.73.3. Consumer Endpoints:

Endpoint	Description
normal	Performs normal search and requires a search query in the search option.
savedsearch	Performs search based on a search query saved in splunk and requires the name of the query in the savedSearch option.

Example

```
from("splunk://normal?
delay=5s&username=user&password=123&initEarliestTime=-10s&search=search index=myindex
sourcetype=someSourcetype")
    .to("direct:search-result");
```

camel-splunk creates a route exchange per search result with a SplunkEvent in the body.

3.73.4. URI Options

Name	Default Value	Context	Description
host	localhost	Both	Splunk host.
port	8089	Both	Splunk port
username	null	Both	Username for Splunk
password	null	Both	Password for Splunk
connectionTimeout	5000	Both	Timeout in MS when connecting to Splunk server
useSunHttpsHandler	false	Both	Use sun.net.www.protocol.https.Handler Https handler to establish the Splunk Connection. Can be useful when running in application servers to avoid app. server https handling.
index	null	Producer	Splunk index to write to
sourceType	null	Producer	Splunk sourcetype argument

Name	Default Value	Context	Description
source	null	Producer	Splunk source argument
tcpReceiverPort	0	Producer	Splunk tcp receiver port when using tcp producer endpoint.
initEarliestTime	null	Consumer	Initial start offset of the first search. Required
earliestTime	null	Consumer	Earliest time of the search time window.
latestTime	null	Consumer	Latest time of the search time window.
count	0	Consumer	A number that indicates the maximum number of entities to return. Note this is not the same as maxMessagesPerPoll which currently is unsupported
search	null	Consumer	The Splunk query to run
savedSearch	null	Consumer	The name of the query saved in Splunk to run
streaming	false	Consumer	Camel 2.14.0 : Stream exchanges as they are received from Splunk, rather than returning all of them in one batch. This has the benefit of receiving results faster, as well as requiring less memory as exchanges aren't buffered in the component.

3.73.5. Message body

Splunk operates on data in key/value pairs. The SplunkEvent class is a placeholder for such data, and should be in the message body

for the producer. Likewise it will be returned in the body per search result for the consumer.

3.73.6. Use Cases

Search Twitter for tweets with music and publish events to Splunk

```
from("twitter://search?type=polling&keywords=music&delay=10&consumerKey=
abc&consumerSecret=def&accessToken=hi j&accessTokenSecret=xxx")
    .convertBodyTo(SplunkEvent.class)
    .to("splunk://submit?username=foo&password=bar&index=camel-
tweets&sourceType=twitter&source=music-tweets");
```

To convert a Tweet to a SplunkEvent you could use a converter like

```
@Converter
public class Tweet2SplunkEvent {
    @Converter
    public static SplunkEvent convertTweet(Status status) {
        SplunkEvent data = new SplunkEvent("twitter-message", null);
        //data.addPair("source", status.getSource());
        data.addPair("from_user", status.getUser().getScreenName());
        data.addPair("in_reply_to", status.getInReplyToScreenName());
        data.addPair(SplunkEvent.COMMON_START_TIME, status.getCreatedAt());
        data.addPair(SplunkEvent.COMMON_EVENT_ID, status.getId());
        data.addPair("text", status.getText());
        data.addPair("retweet_count", status.getRetweetCount());
        if (status.getPlace() != null) {
            data.addPair("place_country", status.getPlace().getCountry());
            data.addPair("place_name", status.getPlace().getName());
            data.addPair("place_street", status.getPlace().getStreetAddress());
        }
        if (status.getGeoLocation() != null) {
            data.addPair("geo_latitude", status.getGeoLocation().getLatitude());
        }
    }
}
```

```

        data.addPair("geo_longitude", status.getGeoLocation().getLongitude());
    }
    return data;
}
}

```

Search Splunk for tweets

```

from("splunk://normal?username=foo&password=bar&initEarliestTime=-2m&search=search
index=camel-tweets sourcetype=twitter")
    .log("${body}");

```

3.73.7. Other comments

Splunk comes with a variety of options for leveraging machine generated data with prebuilt apps for analyzing and displaying this.

For example the jmx app. could be used to publish jmx attributes, eg. route and jvm metrics to Splunk, and displaying this on a dashboard.

3.74. Spring Batch

The **spring-batch** component and support classes provide integration bridge between Camel and [Spring Batch](#) infrastructure.

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-batch</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>

```

3.74.1. URI format

```
spring-batch:jobName[?options]
```

Where **jobName** represents the name of the Spring Batch job located in the Camel registry.

This component can only be used to define producer endpoints, which means that you cannot use the Spring Batch component in a `from()` statement.

3.74.2. Options

Name	Default Value	Description
jobLauncherRef	null	Deprecated and will be removed in Camel 3.0! Camel 2.10: Use <code>jobLauncher=#theName</code> instead.
jobLauncher	null	Camel 2.11.1: Explicitly specifies a JobLauncher to be used from the Camel Registry .

3.74.3. Usage

When Spring Batch component receives the message, it triggers the job execution. The job will be executed using the `org.springframework.batch.core.launch.JobLauncher` instance resolved according to the following algorithm:

- if `JobLauncher` is manually set on the component, then use it.
- if `jobLauncherRef` option is set on the component, then search Camel [Registry](#) for the `JobLauncher` with the given name. **Deprecated and will be removed in Camel 3.0!**
- if there is `JobLauncher` registered in the Camel [Registry](#) under **jobLauncher** name, then use it.
- if none of the steps above allow to resolve the `JobLauncher` and there is exactly one `JobLauncher` instance in the Camel [Registry](#), then use it.

All headers found in the message are passed to the `JobLauncher` as job parameters. String, Long, Double and `java.util.Date` values are copied to the `org.springframework.batch.core.JobParametersBuilder` - other data types are converted to Strings.

3.74.4. Examples

Triggering the Spring Batch job execution:

```
from("direct:startBatch").to("spring-batch:myJob");
```

Triggering the Spring Batch job execution with the `JobLauncher` set explicitly.

```
from("direct:startBatch").to("spring-batch:myJob?jobLauncherRef=myJobLauncher");
```

Starting from the Camel **2.11.1** `JobExecution` instance returned by the `JobLauncher` is forwarded by the `SpringBatchProducer` as the output message. You can use the `JobExecution` instance to perform some operations using the Spring Batch API directly.

```
from("direct:startBatch").to("spring-batch:myJob").to("mock:JobExecutions");
...
MockEndpoint mockEndpoint = ...;
JobExecution jobExecution =
mockEndpoint.getExchanges().get(0).getIn().getBody(JobExecution.class);
BatchStatus currentJobStatus = jobExecution.getStatus();
```

3.74.5. Support classes

Apart from the Component, Camel Spring Batch provides also support classes, which can be used to hook into Spring Batch infrastructure.

3.74.5.1. CamelItemReader

`CamelItemReader` can be used to read batch data directly from the Camel infrastructure.

For example the snippet below configures Spring Batch to read data from JMS queue.

```
<bean id="camelReader"
class="org.apache.camel.component.spring.batch.support.CamelItemReader">
  <constructor-arg ref="consumerTemplate"/>
```

```

    <constructor-arg value="jms:dataQueue" />
</bean>

<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="camelReader" writer="someWriter" commit-interval="100" />
    </batch:tasklet>
  </batch:step>
</batch:job>

```

3.74.5.2. CamelItemWriter

CamelItemWriter has similar purpose as CamelItemReader, but it is dedicated to write chunk of the processed data.

For example the snippet below configures Spring Batch to read data from JMS queue.

```

<bean id="camelwriter"
class="org.apache.camel.component.spring.batch.support.CamelItemWriter">
  <constructor-arg ref="producerTemplate" />
  <constructor-arg value="jms:dataQueue" />
</bean>

<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>
      <batch:chunk reader="someReader" writer="camelwriter" commit-interval="100" />
    </batch:tasklet>
  </batch:step>
</batch:job>

```

3.74.5.3. CamelItemProcessor

CamelItemProcessor is the implementation of Spring Batch `org.springframework.batch.item.ItemProcessor` interface. The latter implementation relays on [Request Reply pattern](#) to delegate the processing of the batch item to the Camel infrastructure. The item to process is sent to the Camel endpoint as the body of the message.

For example the snippet below performs simple processing of the batch item using the [Direct endpoint](#) and the [Simple expression language](#).

```

<camel:camelContext>
  <camel:route>
    <camel:from uri="direct:processor" />
    <camel:setExchangePattern pattern="InOut" />
    <camel:setBody>
      <camel:simple>Processed ${body}</camel:simple>
    </camel:setBody>
  </camel:route>
</camel:camelContext>

<bean id="camelProcessor"
class="org.apache.camel.component.spring.batch.support.CamelItemProcessor">
  <constructor-arg ref="producerTemplate" />
  <constructor-arg value="direct:processor" />
</bean>

<batch:job id="myJob">
  <batch:step id="step">
    <batch:tasklet>

```

```

        <batch:chunk reader="someReader" writer="someWriter"
processor="camelProcessor" commit-interval="100"/>
    </batch:tasklet>
</batch:step>
</batch:job>

```

3.74.5.4. CamelJobExecutionListener

`CamelJobExecutionListener` is the implementation of the `org.springframework.batch.core.JobExecutionListener` interface sending job execution events to the Camel endpoint.

The `org.springframework.batch.core.JobExecution` instance produced by the Spring Batch is sent as a body of the message. To distinguish between before- and after-callbacks `SPRING_BATCH_JOB_EVENT_TYPE` header is set to the `BEFORE` or `AFTER` value.

The example snippet below sends Spring Batch job execution events to the JMS queue.

```

<bean id="camelJobExecutionListener"
class="org.apache.camel.component.spring.batch.support.CamelJobExecutionListener">
    <constructor-arg ref="producerTemplate"/>
    <constructor-arg value="jms:batchEventsBus"/>
</bean>

<batch:job id="myJob">
    <batch:step id="step">
        <batch:tasklet>
            <batch:chunk reader="someReader" writer="someWriter" commit-interval="100"/>
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="camelJobExecutionListener"/>
    </batch:listeners>
</batch:job>

```

3.75. Spring Event

The **spring-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 *Message Filter*.

3.75.1. URI format

```
spring-event://default[?options]
```

Note, at the moment there are no options for this component. That can easily change in future releases, so please check back.

3.76. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.76.1. 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&...`

3.76.2. Options

Name	Type	Description
<code>inputChannel</code>	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.
<code>outputChannel</code>	String	The Spring integration output channel name that is used to send messages to the Spring integration context.
<code>inOut</code>	String	The exchange pattern that the Spring integration endpoint should use. If <code>inOut=true</code> then a reply channel is expected, either from the Spring Integration Message header or configured on the endpoint.

3.76.3. 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.

3.76.4. Examples

3.76.4.1. 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"/>

<!-- custom 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>
```

3.76.4.2. 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"/>

```

3.77. Spring LDAP

Available since Camel 2.11

The **spring-ldap** component provides a Camel wrapper for [Spring LDAP](#).

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-ldap</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>

```

3.77.1. URI format

```
spring-ldap:springLdapTemplate[?options]
```

Where **springLdapTemplate** is the name of the [Spring LDAP Template bean](#). In this bean, you configure the URL and the credentials for your LDAP access.

3.77.2. Options

Name	Type	Description
operation	String	The LDAP operation to be performed. Must be one of search, bind, or unbind.
scope	String	The scope of the search operation. Must be one of object, onelevel, or subtree, see also http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol#Search_and_Compare

If an unsupported value is specified for some option, the component throws an `UnsupportedOperationException`.

3.77.3. Usage

The component supports producer endpoint only. An attempt to create a consumer endpoint will result in an `UnsupportedOperationException`.

The body of the message must be a map (an instance of `java.util.Map`). This map must contain at least an entry with the key **dn** that specifies the root node for the LDAP operation to be performed. Other entries of the map are operation-specific (see below).

The body of the message remains unchanged for the bind and unbind operations. For the search operation, the body is set to the result of

the search, see <http://static.springsource.org/spring-ldap/site/apidocs/org/springframework/ldap/core/LdapTemplate.html#search%28java.lang.String,%20java.lang.String,%20int,%20org.springframework.ldap.core.AttributesMapper%29>.

3.77.3.1. Search

The message body must have an entry with the key **filter**. The value must be a String representing a valid LDAP filter, see http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol#Search_and_Compare.

3.77.3.2. Bind

The message body must have an entry with the key **attributes**. The value must be an instance of [javax.naming.directory.Attributes](#). This entry specifies the LDAP node to be created.

3.77.3.3. Unbind

No further entries necessary, the node with the specified **dn** is deleted.

Key definitions

In order to avoid spelling errors, the following constants are defined in `org.apache.camel.springldap.SpringLdapProducer`:

- `public static final String DN = "dn"`
- `public static final String FILTER = "filter"`
- `public static final String ATTRIBUTES = "attributes"`

3.78. Spring Redis

Available as of Camel 2.11

This component allows sending and receiving messages from [Redis](#). Redis is advanced key-value store where keys can contain strings, hashes, lists, sets and sorted sets. In addition it provides pub/sub functionality for inter-app communications.

Camel provides a producer for executing commands, consumer for subscribing to pub/sub messages an idempotent repository for filtering out duplicate messages.

Prerequisites:

In order to use this component, you must have a Redis server running.

3.78.1. URI Format

```
spring-redis://host:port[?options]
```

You can append query options to the URI in the following format, `?options=value&option2=value&...`

3.78.2. URI Options

Name	Default Value	Context	Description
host	null	Both	The host where Redis server is running.
port	null	Both	Redis port number.
command	SET	Both	Default command, which can be overridden by message header.
channels	SET	Consumer	List of topic names or name patterns to subscribe to.
redisTemplate	null	Producer	Reference to a pre-configured <code>org.springframework.data.redis.core.RedisTemplate</code> instance in the Registry.
connectionFactory	null	Both	Reference to a <code>org.springframework.data.redis.connection.RedisConnectionFactory</code> instance in the Registry.
listenerContainer	null	Consumer	Reference to a <code>org.springframework.data.redis.listener.RedisMessageListenerContainer</code> instance in the Registry.
serializer	null	Consumer	Reference to a <code>org.springframework.data.redis.serializer.RedisSerializer</code> instance in the Registry.

3.78.3. Usage

3.78.3.1. Message headers evaluated by the Redis producer

The producer issues commands to the server and each command has different set of parameters with specific types. The result from the command execution is returned in the message body.

Hash Commands	Description	Parameters	Result
HSET	Set the string value of a hash field	<code>CamelRedis.Key (String)</code> , <code>CamelRedis.Field (String)</code> , <code>CamelRedis.Value (Object)</code>	void
HGET	Get the value of a hash field	<code>CamelRedis.Key (String)</code> , <code>CamelRedis.Field (String)</code>	String
HSETNX	Set the value of a hash field, only if the field does not exist	<code>CamelRedis.Key (String)</code> , <code>CamelRedis.Field (String)</code> , <code>CamelRedis.Value (Object)</code>	void
HMSET	Set multiple hash fields to multiple values	<code>CamelRedis.Key (String)</code> , <code>CamelRedis.Values(Map<String, Object>)</code>	void
HMGET	Get the values of all the given hash fields	<code>CamelRedis.Key (String)</code> , <code>CamelRedis.Fields (Collection<String>)</code>	<code>Collection<Object></code>
HINCRBY	Increment the integer value of a hash field by the given number	<code>CamelRedis.Key (String)</code> , <code>CamelRedis.Field (String)</code> , <code>CamelRedis.Value (Long)</code>	Long
HEXISTS	Determine if a hash field exists	<code>CamelRedis.Key (String)</code> , <code>CamelRedis.Field (String)</code>	Boolean
HDEL	Delete one or more hash fields	<code>CamelRedis.Key (String)</code> , <code>CamelRedis.Field (String)</code>	void
HLEN	Get the number of fields in a hash	<code>CamelRedis.Key (String)</code>	Long
HKEYS	Get all the fields in a hash	<code>CamelRedis.Key (String)</code>	<code>Set<String></code>
HVALS	Get all the values in a hash	<code>CamelRedis.Key (String)</code>	<code>Collection<Object></code>
HGETALL	Get all the fields and values in a hash	<code>CamelRedis.Key (String)</code>	<code>Map<String, Object></code>

List Commands	Description	Parameters	Result
RPUSH	Append one or multiple values to a list	CamelRedis.Key (String), CamelRedis.Value (Object)	Long
RPUSHX	Append a value to a list, only if the list exists	CamelRedis.Key (String), CamelRedis.Value (Object)	Long
LPUSH	Prepend one or multiple values to a list	CamelRedis.Key (String), CamelRedis.Value (Object)	Long
LLEN	Get the length of a list	CamelRedis.Key (String)	Long
LRANGE	Get a range of elements from a list	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long)	List<Object>
LTRIM	Trim a list to the specified range	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long)	void
LINDEX	Get an element from a list by its index	CamelRedis.Key (String), CamelRedis.Index (Long)	String
LINSERT	Insert an element before or after another element in a list	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Pivot (String), CamelRedis.Position (String)	Long
LSET	Set the value of an element in a list by its index	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Index (Long)	void
LREM	Remove elements from a list	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Count (Long)	Long
LPOP	Remove and get the first element in a list	CamelRedis.Key (String)	Object
RPOP	Remove and get the last element in a list	CamelRedis.Key (String)	String
RPOPLPUSH	Remove the last element in a list, append it to another list and return it	CamelRedis.Key (String), CamelRedis.Destination (String)	Object
BRPOPLPUSH	Pop a value from a list, push it to another list and return it; or block until one is available	CamelRedis.Key (String), CamelRedis.Destination (String), CamelRedis.Timeout (Long)	Object
BLPOP	Remove and get the first element in a list, or block until one is available	CamelRedis.Key (String), CamelRedis.Timeout (Long)	Object
BRPOP	Remove and get the last element in a list, or block until one is available	CamelRedis.Key (String), CamelRedis.Timeout (Long)	String

Set Commands	Description	Parameters	Result
SADD	Add one or more members to a set	CamelRedis.Key (String), CamelRedis.Value (Object)	Boolean
SMEMBERS	Get all the members in a set	CamelRedis.Key (String)	Set<Object>
SREM	Remove one or more members from a set	CamelRedis.Key (String), CamelRedis.Value (Object)	Boolean
SPOP	Remove and return a random member from a set	CamelRedis.Key (String)	String
SMOVE	Move a member from one set to another	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Destination (String)	Boolean
SCARD	Get the number of members in a set	CamelRedis.Key (String)	Long
SISMEMBER	Determine if a given value is a member of a set	CamelRedis.Key (String), CamelRedis.Value (Object)	Boolean
SINTER	Intersect multiple sets	CamelRedis.Key (String), CamelRedis.Keys (String)	Set<Object>
SINTERSTORE	Intersect multiple sets and store the resulting set in a key	CamelRedis.Key (String), CamelRedis.Keys (String), CamelRedis.Destination (String)	void

Set Commands	Description	Parameters	Result
SUNION	Add multiple sets	CamelRedis.Key (String), CamelRedis.Keys (String)	Set<Object>
SUNIONSTORE	Add multiple sets and store the resulting set in a key	CamelRedis.Key (String), CamelRedis.Keys (String), CamelRedis.Destination (String)	void
SDIFF	Subtract multiple sets	CamelRedis.Key (String), CamelRedis.Keys (String)	Set<Object>
SDIFFSTORE	Subtract multiple sets and store the resulting set in a key	CamelRedis.Key (String), CamelRedis.Keys (String), CamelRedis.Destination (String)	void
SRANDMEMBER	Get one or multiple random members from a set	CamelRedis.Key (String)	String

Ordered set Commands	Description	Parameters	Result
ZADD	Add one or more members to a sorted set, or update its score if it already exists	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Score (Double)	Boolean
ZRANGE	Return a range of members in a sorted set, by index	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long), CamelRedis.WithScore (Boolean)	Object
ZREM	Remove one or more members from a sorted set	CamelRedis.Key (String), CamelRedis.Value (Object)	Boolean
ZINCRBY	Increment the score of a member in a sorted set	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Increment (Double)	Double
ZRANK	Determine the index of a member in a sorted set	CamelRedis.Key (String), CamelRedis.Value (Object)	Long
ZREVRANK	Determine the index of a member in a sorted set, with scores ordered from high to low	CamelRedis.Key (String), CamelRedis.Value (Object)	Long
ZREVRANGE	Return a range of members in a sorted set, by index, with scores ordered from high to low	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long), CamelRedis.WithScore (Boolean)	Object
ZCARD	Get the number of members in a sorted set	CamelRedis.Key (String),	Long
ZCOUNT	Count the members in a sorted set with scores within the given values	CamelRedis.Key (String), CamelRedis.Min (Double), CamelRedis.Max (Double)	Long
ZRANGEBYSCORE	Return a range of members in a sorted set, by score	CamelRedis.Key (String), CamelRedis.Min (Double), CamelRedis.Max (Double)	Set<Object>
ZREVRANGEBYSCORE	Return a range of members in a sorted set, by score, with scores ordered from high to low	CamelRedis.Key (String), CamelRedis.Min (Double), CamelRedis.Max (Double)	Set<Object>
ZREMRANGEBYRANK	Remove all members in a sorted set within the given indexes	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long)	void
ZREMRANGEBYSCORE	Remove all members in a sorted set within the given scores	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long)	void
ZUNIONSTORE	Add multiple sorted sets and store the resulting sorted set in a new key	CamelRedis.Key (String), CamelRedis.Keys (String), CamelRedis.Destination (String)	void
ZINTERSTORE	Intersect multiple sorted sets and store the resulting sorted set in a new key	CamelRedis.Key (String), CamelRedis.Keys (String), CamelRedis.Destination (String)	void

String Commands	Description	Parameters	Result
SET	Set the string value of a key	CamelRedis.Key (String), CamelRedis.Value (Object)	void
GET	Get the value of a key	CamelRedis.Key (String)	Object
STRLEN	Get the length of the value stored in a key	CamelRedis.Key (String)	Long
APPEND	Append a value to a key	CamelRedis.Key (String), CamelRedis.Value (String)	Integer
SETBIT	Sets or clears the bit at offset in the string value stored at key	CamelRedis.Key (String), CamelRedis.Offset (Long), CamelRedis.Value (Boolean)	void
GETBIT	Returns the bit value at offset in the string value stored at key	CamelRedis.Key (String), CamelRedis.Offset (Long)	Boolean
SETRANGE	Overwrite part of a string at key starting at the specified offset	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Offset (Long)	void
GETRANGE	Get a substring of the string stored at a key	CamelRedis.Key (String), CamelRedis.Start (Long), CamelRedis.End (Long)	String
SETNX	Set the value of a key, only if the key does not exist	CamelRedis.Key (String), CamelRedis.Value (Object)	Boolean
SETEX	Set the value and expiration of a key	CamelRedis.Key (String), CamelRedis.Value (Object), CamelRedis.Timeout (Long), SECONDS	void
DECRBY	Decrement the integer value of a key by the given number	CamelRedis.Key (String), CamelRedis.Value (Long)	Long
DECR	Decrement the integer value of a key by one	CamelRedis.Key (String),	Long
INCRBY	Increment the integer value of a key by the given amount	CamelRedis.Key (String), CamelRedis.Value (Long)	Long
INCR	Increment the integer value of a key by one	CamelRedis.Key (String)	Long
MGET	Get the values of all the given keys	CamelRedis.Fields (Collection<String>)	List<Object>
MSET	Set multiple keys to multiple values	CamelRedis.Values(Map<String, Object>)	void
MSETNX	Set multiple keys to multiple values, only if none of the keys exist	CamelRedis.Key (String), CamelRedis.Value (Object)	void
GETSET	Set the string value of a key and return its old value	CamelRedis.Key (String), CamelRedis.Value (Object)	Object

Key Commands	Description	Parameters	Result
EXISTS	Determine if a key exists	CamelRedis.Key (String)	Boolea
DEL	Delete a key	CamelRedis.Keys (String)	void
TYPE	Determine the type stored at key	CamelRedis.Key (String)	DataType
KEYS	Find all keys matching the given pattern	CamelRedis.Pattern (String)	Collection<String>
RANDOMKEY	Return a random key from the keyspace	CamelRedis.Pattern (String), CamelRedis.Value (String)	String
RENAME	Rename a key	CamelRedis.Key (String)	void
RENAMENX	Rename a key, only if the new key does not exist	CamelRedis.Key (String), CamelRedis.Value (String)	Boolean
EXPIRE	Set a key's time to live in seconds	CamelRedis.Key (String), CamelRedis.Timeout (Long)	Boolean
SORT	Sort the elements in a list, set or sorted set	CamelRedis.Key (String)	List<Object>
PERSIST	Remove the expiration from a key	CamelRedis.Key (String)	Boolean
EXPIREAT	Set the expiration for a key as a UNIX timestamp	CamelRedis.Key (String), CamelRedis.Timestamp (Long)	Boolean
PEXPIRE	Set a key's time to live in milliseconds	CamelRedis.Key (String), CamelRedis.Timeout (Long)	Boolean

Key Commands	Description	Parameters	Result
PEXPIREAT	Set the expiration for a key as a UNIX timestamp specified in milliseconds	CamelRedis.Key (String), CamelRedis.Timestamp (Long)	Boolean
TTL	Get the time to live for a key	CamelRedis.Key (String)	Long
MOVE	Move a key to another database	CamelRedis.Key (String), CamelRedis.Db (Integer)	Boolean

Other Command	Description	Parameters	Result
MULTI	Mark the start of a transaction block	none	void
DISCARD	Discard all commands issued after MULTI	none	void
EXEC	Execute all commands issued after MULTI	none	void
WATCH	Watch the given keys to determine execution of the MULTI/EXEC block	CamelRedis.Keys (String)	void
UNWATCH	Forget about all watched keys	none	void
ECHO	Echo the given string	CamelRedis.Value (String)	String
PING	Ping the server	none	String
QUIT	Close the connection	none	void
PUBLISH	Post a message to a channel	CamelRedis.Channel (String), CamelRedis.Message (Object)	void

3.78.3.2. Redis consumer

The consumer subscribes to a channel either by channel name using `SUBSCRIBE` or a string pattern using `PSUBSCRIBE` commands. When a message is sent to the channel using `PUBLISH` command, it will be consumed and the message will be available as Camel message body. The message is also serialized using configured serializer or the default `JdkSerializationRedisSerializer`.

Message headers set by the Consumer

Header	Type	Description
CamelRedis.Channel	String	The channel name, where the message was received.
CamelRedis.Pattern	String	The pattern matching the channel, where the message was received.

3.78.4. Dependencies

Maven users will need to add the following dependency to their `pom.xml`.

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-redis</artifactId>
  <version>${camel-version}</version>
</dependency>
```

where `${camel-version}` must be replaced by the actual version of Camel (2.11 or higher).

3.79. Spring Web Services

Available as of Camel 2.6

The **spring-ws:** component allows you to integrate with [Spring Web Services](#). It offers both *client*-side support, for accessing web services, and *server*-side support for creating your own contract-first web services.

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-ws</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

Dependencies:

As of Camel 2.8 this component ships with Spring-WS 2.0.x which (like the rest of Camel) requires Spring 3.0.x.

Earlier Camel versions shipped Spring-WS 1.5.9 which is compatible with Spring 2.5.x and 3.0.x. In order to run earlier versions of `camel-spring-ws` on Spring 2.5.x you need to add the `spring-webmvc` module from Spring 2.5.x. In order to run Spring-WS 1.5.9 on Spring 3.0.x you need to exclude the OXM module from Spring 3.0.x as this module is also included in Spring-WS 1.5.9 (see [this post](#))

3.79.1. URI format

The URI scheme for this component is as follows

```
spring-ws:[mapping-type:]address[?options]
```

To expose a web service **mapping-type** needs to be set to any of the following:

Mapping type	Description
rootqname	Offers the option to map web service requests based on the qualified name of the root element contained in the message.
soapaction	Used to map web service requests based on the SOAP action specified in the header of the message.
uri	In order to map web service requests that target a specific URI.
xpathresult	Used to map web service requests based on the evaluation of an XPath expression against the incoming message. The result of the evaluation should match the XPath result specified in the endpoint URI.
beanname	Allows you to reference an <code>org.apache.camel.component</code> . <code>spring.ws.bean.CamelEndpointDispatcher</code> object in order to integrate with existing (legacy) endpoint mappings like <code>PayloadRootQNameEndpointMapping</code> , <code>SoapActionEndpointMapping</code> , etc

As a consumer the **address** should contain a value relevant to the specified mapping-type (e.g. a SOAP action, XPath expression). As a producer the address should be set to the URI of the web service your calling upon.

You can append query **options** to the URI in the following format, `?option=value&option=value&...`

3.79.2. Options

Name	Required?	Description
soapAction	No	SOAP action to include inside a SOAP request when accessing remote web services
wsAddressingAction	No	WS-Addressing 1.0 action header to include when accessing web services. The <code>To</code> header is set to the <i>address</i> of the web service as specified in the endpoint URI (default Spring-WS behavior).
expression	Only when <i>mapping-type</i> is <code>xpathresult</code>	XPath expression to use in the process of mapping web service requests, should match the result specified by <code>xpathresult</code>

Name	Required?	Description
timeout	No	<p>Camel 2.10: Sets the socket read timeout (in milliseconds) while invoking a webservice using the producer, see URLConnection.setReadTimeout() and CommonsHttpClient.setReadTimeout(). This option works when using the built-in message sender implementations: <i>CommonsHttpClient</i> and <i>URLConnectionMessageSender</i>. One of these implementations will be used by default for HTTP based services unless you customize the Spring WS configuration options supplied to the component. If you are using a non-standard sender, it is assumed that you will handle your own timeout configuration.</p> <p>Camel 2.12: The built-in message sender <i>HttpComponentsMessageSender</i> is considered instead of <i>CommonsHttpClient</i> which has been deprecated, see HttpComponentsMessageSender.setReadTimeout().</p>
sslContextParameters	No	<p>Camel 2.10: Reference to an <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry. See Using the JSSE Configuration Utility. This option works when using the built-in message sender implementations: <i>CommonsHttpClient</i> and <i>URLConnectionMessageSender</i>. One of these implementations will be used by default for HTTP based services unless you customize the Spring WS configuration options supplied to the component. If you are using a non-standard sender, it is assumed that you will handle your own TLS configuration.</p> <p>Camel 2.12: The built-in message sender <i>HttpComponentsMessageSender</i> is considered instead of <i>CommonsHttpClient</i> which has been deprecated.</p>

3.79.2.1. Registry based options

The following options can be specified in the registry (most likely a Spring ApplicationContext) and referenced from the endpoint URI using the # notation.

Name	Required?	Description
webServiceTemplate	No	Option to provide a custom WebServiceTemplate . This allows for full control over client-side web services handling; like adding a custom interceptor or specifying a fault resolver, message sender or message factory.
messageSender	No	Option to provide a custom WebServiceMessageSender . For example to perform authentication or use alternative transports
messageFactory	No	Option to provide a custom WebServiceMessageFactory . For example when you want Apache Axiom to handle web service messages instead of SAAJ
transformerFactory	No	Option to override default TransformerFactory. The provided transformer factory must be of type <code>javax.xml.transform.TransformerFactory</code>
endpointMapping	Only when <i>mapping-type</i> is <i>rootname</i> , <i>soapaction</i> , <i>uri</i> or <i>xpathresult</i>	<p>Reference to an instance of <code>org.apache.camel.component.spring.ws.Bean.CamelEndpointMapping</code> in the Registry/ApplicationContext. Only one bean is required in the registry to serve all Camel/Spring-WS endpoints. This bean is auto-discovered by the MessageDispatcher and used to map requests to Camel endpoints based on characteristics specified on the endpoint (like root QName, SOAP action, etc)</p>

Name	Required?	Description
messageFilter	No	Camel 2.10.3 Option to provide a custom MessageFilter. For example when you want to process your headers or attachments by your own.

3.79.3. Message headers

Name	Type	Description
CamelSpringWebserviceEndpointUri	String	URI of the web service your accessing as a client, overrides <i>address</i> part of the endpoint URI
CamelSpringWebserviceSoapAction	String	Header to specify the SOAP action of the message, overrides <i>soapAction</i> option if present
CamelSpringWebserviceAddressingAction	URI	Use this header to specify the WS-Addressing action of the message, overrides <i>wsAddressingAction</i> option if present
CamelSpringWebserviceSoapHeader	Source	Camel 2.11.1: Use this header to specify/access the SOAP headers of the message.

3.79.4. Accessing web services

To call a web service at `http://foo.com/bar` simply define a route:

```
from("direct:example").to("spring-ws:http://foo.com/bar")
```

And sent a message:

```
template.requestBody("direct:example", "<foobar  
xmlns=\"http://foo.com\"><msg>test message</msg></foobar>");
```

Remember if it's a SOAP service you're calling you don't have to include SOAP tags. Spring-WS will perform the XML-to-SOAP marshaling.

3.79.4.1. Sending SOAP and WS-Addressing action headers

When a remote web service requires a SOAP action or use of the WS-Addressing standard you define your route as:

```
from("direct:example")  
.to("spring-ws:http://foo.com/bar?soapAction=http://foo.com&wsAddressingAction=http://bar.com")
```

Optionally you can override the endpoint options with header values:

```
template.requestBodyAndHeader("direct:example",  
"<foobar xmlns=\"http://foo.com\"><msg>test message</msg></foobar>",  
SpringWebserviceConstants.SPRING_WS_SOAP_ACTION, "http://baz.com");
```

3.79.4.2. Using SOAP headers

Available as of Camel 2.11.1

You can provide the SOAP header(s) as a Camel Message header when sending a message to a spring-ws endpoint, for example given the following SOAP header in a String

```
String body = ...
String soapHeader = "<h:Header
xmlns:h=\"http://www.webserviceX.NET/\"><h:MessageID>1234567890</h:MessageID><h:Nes
ted><h:NestedID>1111</h:NestedID></h:Nested></h:Header>";
```

We can set the body and header on the Camel Message as follows:

```
exchange.getIn().setBody(body);
exchange.getIn().setHeader(SpringWebserviceConstants.SPRING_WS_SOAP_HEADER, soapHeader);
```

And then send the Exchange to a spring-ws endpoint to call the Web Service.

Likewise the spring-ws consumer will also enrich the Camel Message with the SOAP header.

For an example see this [unit test](#).

3.79.4.3. The header and attachment propagation

Spring WS Camel supports propagation of the headers and attachments into Spring-WS WebServiceMessage response since version **2.10.3**. The endpoint will use so called "hook" the MessageFilter (default implementation is provided by BasicMessageFilter) to propagate the exchange headers and attachments into WebServiceMessage response. Now you can use

```
exchange.getOut().getHeaders().put("myCustom", "myHeaderValue")
exchange.getIn().addAttachment("myAttachment", new DataHandler(...))
```

Note: If the exchange header in the pipeline contains text, it generates QName(key)=value attribute in the soap header. Recommended is to create a QName class directly and put into any key into header.

3.79.4.4. How to use MTOM attachments

The BasicMessageFilter provides all required information for Apache Axiom in order to produce MTOM message. If you want to use Apache Camel Spring WS within Apache Axiom, here is an example:

1. Simply define the messageFactory as is bellow and Spring-WS will use MTOM strategy to populate your SOAP message with optimized attachments.

```
<bean id="axiomMessageFactory"
class="org.springframework.ws.soap.axiom.AxiomSoapMessageFactory">
<property name="payloadCaching" value="false" />
<property name="attachmentCaching" value="true" />
<property name="attachmentCacheThreshold" value="1024" />
</bean>
```

2. Add into your pom.xml the following dependencies

```
<dependency>
<groupId>org.apache.ws.commons.axiom</groupId>
<artifactId>axiom-api</artifactId>
<version>1.2.13</version>
</dependency>
<dependency>
<groupId>org.apache.ws.commons.axiom</groupId>
<artifactId>axiom-impl</artifactId>
<version>1.2.13</version>
<scope>runtime</scope>
```

```
</dependency>
```

3. Add your attachment into the pipeline, for example using a Processor implementation.

```
private class Attachement implements Processor {
public void process(Exchange exchange) throws Exception
{ exchange.getOut().copyFrom(exchange.getIn()); File file = new
File("testAttachment.txt"); exchange.getOut().addAttachment("test", new
DataHandler(new FileDataSource(file))); }
}
```

4. Define endpoint (producer) as usual, for example like this:

```
from("direct:send")
.process(new Attachement())
.to("spring-ws:http://localhost:8089/mySoapService?soapAction=mySoap&messageFactory
=axiomMessageFactory");
```

5. Now, your producer will generate MTOM message with optimized attachments.

3.79.4.5. The custom header and attachment filtering

If you need to provide your custom processing of either headers or attachments, extend existing `BasicMessageFilter` and override the appropriate methods or write a brand new implementation of the `MessageFilter` interface.

To use your custom filter, add this into your spring context:

You can specify either a global a or a local message filter as follows:

a) the global custom filter that provides the global configuration for all Spring-WS endpoints

```
<bean id="messageFilter" class="your.domain.myMessageFiler" scope="singleton" />
```

or

b) the local messageFilter directly on the endpoint as follows:

```
to("spring-ws:http://yourdomain.com?messageFilter=#myEndpointSpecificMessageFilter");
```

For more information see [CAMEL-5724](#)

If you want to create your own `MessageFilter`, consider overriding the following methods in the default implementation of `MessageFilter` in class `BasicMessageFilter`:

```
protected void doProcessSoapHeader(Message inOrOut, SoapMessage soapMessage)
{ your code /*no need to call super*/ }

protected void doProcessSoapAttachments(Message inOrOut, SoapMessage response)
{ your code /*no need to call super*/ }
```

3.79.4.6. Using a custom MessageSender and MessageFactory

A custom message sender or factory in the registry can be referenced like this:

```
from("direct:example")
.to("spring-ws:http://foo.com/bar?messageFactory=#messageFactory&messageSender=#mes
sageSender")
```

Spring configuration:

```

<!-- authenticate using HTTP Basic Authentication -->
<bean id="messageSender"
class="org.springframework.ws.transport.http.HttpComponentsMessageSender">
    <property name="credentials">
        <bean class="org.apache.commons.httpclient.UsernamePasswordCredentials">
            <constructor-arg index="0" value="admin"/>
            <constructor-arg index="1" value="secret"/>
        </bean>
    </property>
</bean>

<!-- force use of Sun SAAJ implementation,
http://static.springsource.org/spring-ws/sites/1.5/faq.html#saa-jboss -->
<bean id="messageFactory"
class="org.springframework.ws.soap.saa.SaaJSoapMessageFactory">
    <property name="messageFactory">
        <bean
            class="com.sun.xml.messaging.saa.soap.ver1_1.SOAPMessageFactory1_1Impl"></
        bean>
    </property>
</bean>

```

3.79.5. Exposing web services

In order to expose a web service using this component you first need to set-up a [MessageDispatcher](#) to look for endpoint mappings in a Spring XML file. If you plan on running inside a servlet container you probably want to use a `MessageDispatcherServlet` configured in `web.xml`.

By default the `MessageDispatcherServlet` will look for a Spring XML named `/WEB-INF/spring-ws-servlet.xml`. To use Camel with Spring-WS the only mandatory bean in that XML file is `CamelEndpointMapping`. This bean allows the `MessageDispatcher` to dispatch web service requests to your routes.

web.xml

```

<web-app>
    <servlet>
        <servlet-name>spring-ws</servlet-name>
        <servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring-ws</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>

```

spring-ws-servlet.xml

```

<bean id="endpointMapping"
class="org.apache.camel.component.spring.ws.bean.CamelEndpointMapping" />

<bean id="wsdl" class="org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition">
    <property name="schema">
        <bean class="org.springframework.xml.xsd.SimpleXsdSchema">
            <property name="xsd" value="/WEB-INF/foobar.xsd"/>
        </bean>
    </property>
    <property name="portTypeName" value="FooBar" />
    <property name="locationUri" value="/" />
    <property name="targetNamespace" value="http://example.com/" />
</bean>

```

More information on setting up Spring-WS can be found in [Writing Contract-First Web Services](#). Basically paragraph 3.6 "Implementing the Endpoint" is handled by this component (specifically paragraph 3.6.2 "Routing the Message to the Endpoint" is where `CamelEndpointMapping` comes in). Also don't forget to check out the [Spring Web Services Example](#) included in the Camel distribution.

3.79.5.1. Endpoint mapping in routes

With the XML configuration in-place you can now use Camel's DSL to define what web service requests are handled by your endpoint:

The following route will receive all web service requests that have a root element named "GetFoo" within the `http://example.com/ namespace`.

```
from( "spring-ws:rootqname:{http://example.com/}GetFoo?endpointMapping=#endpointMapping" )
    .convertBodyTo( String.class ).to( mock:example )
```

The following route will receive web service requests containing the `http://example.com/GetFoo SOAP action`.

```
from( "spring-ws:soapaction:http://example.com/GetFoo?endpointMapping=#endpointMapping" )
    .convertBodyTo( String.class ).to( mock:example )
```

The following route will receive all requests sent to `http://example.com/foobar`.

```
from( "spring-ws:uri:http://example.com/foobar?endpointMapping=#endpointMapping" )
    .convertBodyTo( String.class ).to( mock:example )
```

The route below will receive requests that contain the element `<foobar>abc</foobar>` anywhere inside the message (and the default namespace).

```
from( "spring-ws:xpathresult:abc?expression=//foobar&endpointMapping=#endpointMapping" )
    .convertBodyTo( String.class ).to( mock:example )
```

3.79.5.2. Alternative configuration, using existing endpoint mappings

For every endpoint with mapping-type `beanname` one bean of type `CamelEndpointDispatcher` with a corresponding name is required in the Registry/Application Context. This bean acts as a bridge between the Camel endpoint and an existing [endpoint mapping](#) like `PayloadRootQNameEndpointMapping`.

The use of the `beanname` mapping-type is primarily meant for (legacy) situations where you're already using Spring-WS and have endpoint mappings defined in a Spring XML file. The `beanname` mapping-type allows you to wire your Camel route into an existing endpoint mapping. When you're starting from scratch it's recommended to define your endpoint mappings as Camel URI's (as illustrated above with `endpointMapping`) since it requires less configuration and is more expressive. Alternatively you could use vanilla Spring-WS with the help of annotations.

An example of a route using `beanname`:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="spring-ws:beanname:QuoteEndpointDispatcher" />
    <to uri="mock:example" />
  </route>
</camelContext>

<bean id="legacyEndpointMapping"
```

```

class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">
  <property name="mappings">
    <props>
      <prop
        key="{http://example.com/}GetFuture">FutureEndpointDispatcher</prop>
      <prop key="{http://example.com/}GetQuote">QuoteEndpointDispatcher</prop>
    </props>
  </property>
</bean>

<bean id="QuoteEndpointDispatcher"
class="org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher" />
<bean id="FutureEndpointDispatcher"
class="org.apache.camel.component.spring.ws.bean.CamelEndpointDispatcher" />

```

3.79.6. POJO (un)marshalling

Camel's [pluggable data formats](#) offer support for pojo/xml marshalling using libraries such as JAXB, XStream, JibX, Castor and XMLBeans. You can use these data formats in your route to sent and receive pojo's, to and from web services.

When *accessing* web services you can marshal the request and unmarshal the response message:

```

JaxbDataFormat jaxb = new JaxbDataFormat(false);
jaxb.setContextPath("com.example.model");

from("direct:example").marshal(jaxb).to("spring-ws:http://foo.com/bar").unmarshal(j
axb);

```

Similarly when *providing* web services, you can unmarshal XML requests to POJO's and marshal the response message back to XML:

```

from("spring-ws:rootqname:{http://example.com/}GetFoo?endpointMapp
ing=#endpointMapping").unmarshal(jaxb)
.to("mock:example").marshal(jaxb);

```

3.80. 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.

3.80.1. 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)
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 <code>javax.security.auth.Subject</code> into a Spring Security Authentication instance.
useThreadSecurity-Context	true	If a <code>javax.security.auth.Subject</code> cannot be found in the In message header under <code>Exchange.AUTHENTICATION</code> , check the Spring Security <code>SecurityContextHolder</code> for an Authentication object.
always-Reauthenticate	false	If set to true, the <code>SpringSecurityAuthorizationPolicy</code> will always call <code>AuthenticationManager.authenticate()</code> each time the policy is accessed.

3.80.2. 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:

```
<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`.

3.80.3. 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(authToken);

// 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 [section 5.3.1](#) 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.

3.80.4. 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>

```

3.80.5. 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`.

3.81. SQL Component

The **sql** component allows you to work with databases using JDBC queries. The difference between this component and *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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

The SQL component also supports:

- a JDBC based repository for the *Idempotent Consumer* EIP pattern. See further below.
- a JDBC based repository for the *Aggregator* EIP pattern. See further below.

In Camel 2.10 or older the SQL component can be used only as a producer. Starting with Camel 2.11, however, you will be able to use this as a consumer, i.e., define within a `from()` statement.

3.81.1. URI format

From Camel 2.11 onwards this component can create both consumer (e.g. `from()`) and producer endpoints (e.g. `to()`). In previous versions, it could only act as a producer.

This component can be used as a *Transactional Client*.

The SQL component uses the following endpoint URI notation:

```
sql:select * from table where id=# order by name[?options]
```

From Camel 2.11 onwards you can use named parameters by using `#:name` style as shown:

```
sql:select * from table where id=:myId order by name[?options]
```

When using named parameters, Camel will lookup the names from, in the given precedence:

1. from message body if its a `java.util.Map`
2. from message headers

If a named parameter cannot be resolved, then an exception is thrown.

From Camel 2.14 onward you can use Simple expressions as parameters as shown:

```
sql:select * from table where id=:${property.myId} 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&...

3.81.2. Options

Option	Type	Default	Description
batch	boolean	false	Camel 2.7.5, 2.8.4 and 2.9: Execute SQL batch update statements. See notes below on how the treatment of the inbound message body changes if this is set to true.
dataSourceRef	String	null	Deprecated and will be removed in Camel 3.0: Reference to a <code>DataSource</code> to look up in the registry. Use <code>dataSource=#theName</code> instead.
dataSource	String	null	Starting with Camel 2.11, Reference to a <code>DataSource</code> to look up in the registry.
placeholder	String	#	Specifies a character that will be replaced to ? in SQL query. Note that it is a simple <code>String.replaceAll()</code> operation and no SQL parsing is involved (quoted strings will also change). This replacement happens only if the endpoint is created using the <code>SqlComponent</code> . If you manually create the endpoint use the expected ? sign instead.
template.<xxx>		null	Sets additional options on the Spring <code>JdbcTemplate</code> that is used behind the scenes to execute the queries. For instance, <code>template.maxRows=10</code> . For detailed documentation, see the JdbcTemplate javadoc documentation.
allowNamed-Parameters	boolean	true	Camel 2.11: Whether to allow using named parameters in the queries.
processing-Strategy			Camel 2.11: SQL consumer only: Allows for plugging in a custom <code>org.apache.camel.component.sql.SqlProcessingStrategy</code> to execute queries when the consumer has processed the rows/batch.
prepareStatement-Strategy			Camel 2.11: Allows to plugin to use a custom <code>org.apache.camel.component.sql.SqlPreparedStatementStrategy</code> to control preparation of the query and prepared statement.
consumer.delay	long	500	Camel 2.11: SQL consumer only: Delay in milliseconds between each poll.
consumer.initial-Delay	long	1000	Camel 2.11: SQL consumer only: Milliseconds before polling starts.
consumer.useFixedDelay	boolean	false	Camel 2.11: SQL consumer only: Set to true to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in the Java doc for details.
maxMessagesPerPoll	int	0	Camel 2.11: (SQL consumer only) An integer value to define the maximum number of messages to gather per poll. By default, no maximum is set.
consumer.useIterator	boolean	true	Camel 2.11: (SQL consumer only): If true each row returned when polling will be processed individually. If false the entire <code>java.util.List</code> of data is set as the IN body.
consumer.route-EmptyResultSet	boolean	false	Camel 2.11: SQL consumer only: Whether to route a single empty Exchange if there was no data to poll.
consumer.onConsume	String	null	Camel 2.11: SQL consumer only: After processing each row then this query can be executed, if the Exchange was processed

Option	Type	Default	Description
			successfully, for example to mark the row as processed. The query can have parameter.
<code>consumer.onConsume-Failed</code>	String	null	Camel 2.11: (SQL consumer only) After processing each row this query can be executed, if the Exchange failed, for example to mark the row as failed. The query can take parameters.
<code>consumer.onConsume-BatchComplete</code>	String	null	Camel 2.11: (SQL consumer only) After processing the entire batch, this query can be executed to bulk update rows etc. The query cannot have parameters.
<code>consumer.expected-UpdateCount</code>	int	-1	Camel 2.11: (SQL consumer only) If using <code>consumer.onConsume</code> then this option can be used to set an expected number of rows being updated. Typically you may set this to 1 to expect one row to be updated.
<code>consumer.break-BatchOnConsumeFail</code>	boolean	false	Camel 2.11: (SQL consumer only) If using <code>consumer.onConsume</code> and it fails, then this option controls whether to break out of the batch or continue processing the next row from the batch.
<code>alwaysPopulate-Statement</code>	boolean	false	Camel 2.11: (SQL producer only) If enabled then the <code>populateStatement</code> method from <code>org.apache.camel.component.sql.SqlPrepareStatementStrategy</code> is always invoked, also if there are no expected parameters to be prepared. If set to false then the <code>populateStatement</code> is only invoked if there are one or more expected parameters to be set; for example this avoids reading the message body/headers for SQL queries with no parameters.
<code>separator</code>	char	,	Camel 2.11.1: The separator to use when parameter values is taken from message body (if the body is a String type), to be inserted at # placeholders. Notice if you use named parameters, then a Map type is used instead.
<code>outputType</code>	String	SelectList	Camel 2.12.0: Make the output of consumer or producer to <code>SelectList</code> as List of Map, or <code>SelectOne</code> as single Java object in the following way: a) If the query has only single column, then that JDBC Column object is returned. (such as <code>SELECT COUNT(*) FROM PROJECT</code> will return a Long object. b) If the query has more than one column, then it will return a Map of that result. c) If the <code>outputClass</code> is set, then it will convert the query result into an Java bean object by calling all the setters that match the column names. It will assume your class has a default constructor to create instance with. d) If the query resulted in more than one rows, it throws a non-unique result exception.
<code>outputClass</code>	String	null	Camel 2.12.0: Specify the full package and class name to use as conversion when <code>outputType=SelectOne</code> .
<code>parametersCount</code>	int	0	Camel 2.11.2/2.12.0: If set greater than zero, then Camel will use this count value of parameters to replace instead of querying via JDBC metadata API. This is useful if the JDBC vendor could not return correct parameters count, then user may override instead.
<code>noop</code>	boolean	false	Camel 2.12.0: If set, will ignore the results of the SQL query and use the existing IN message as the OUT message for the continuation of processing

3.81.3. 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.

3.81.4. 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`.

3.81.5. Header values

When performing `update` operations, the SQL Component stores the update count in the following message headers:

Header	Description
<code>CamelSqlUpdateCount</code>	The number of rows updated for <code>update</code> operations, returned as an <code>Integer</code> object.
<code>CamelSqlRowCount</code>	The number of rows returned for <code>select</code> operations, returned as an <code>Integer</code> object.
<code>CamelSqlQuery</code>	Query to execute. This query takes precedence over the query specified in the endpoint URI. Note that query parameters in the header are represented by a <code>?</code> instead of a <code>#</code> symbol.

When performing `insert` operations, the SQL Component stores the rows with the generated keys and number of these rows in the following message headers (**Available as of Camel 2.12.4, 2.13.1**):

Header	Description
<code>CamelSqlGeneratedKeysRowCount</code>	The number of rows in the header that contains generated keys.
<code>CamelSqlGeneratedKeyRows</code>	Rows that contains the generated keys (a list of maps of keys).

3.81.6. Generated keys

Available as of Camel 2.12.4, 2.13.1 and 2.14

If you insert data using SQL `INSERT`, then the RDBMS may support auto generated keys. You can instruct the SQL producer to return the generated keys in headers.

To do that set the header `CamelSqlRetrieveGeneratedKeys=true`. Then the generated keys will be provided as headers with the keys listed in the table above.

You can see more details in this [unit test](#).

3.81.7. Configuration

A reference to a `DataSource` can be set in the URI as shown:

```
sql:select * from table where id=# order by name?dataSource=myDS
```

3.81.8. 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 = assertInstanceOf(
    List.class, mock.getReceivedExchanges().get(0).getIn().getBody());

// and each row in the list is a Map
Map row = assertInstanceOf(Map.class, received.get(0));

// and we should be able to 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"/>
```

3.81.9. Using named parameters

This feature is available starting with Camel 2.11.

In the given route below, we want to get all the projects from the projects table. Notice the SQL query has 2 named parameters, `:#lic` and `:#min`.

Camel will then lookup for these parameters from the message body or message headers. Notice in the example above we set two headers with constant value for the named parameters:

```
from("direct:projects")
  .setHeader("lic", constant("ASF"))
  .setHeader("min", constant(123))
  .to("sql:select * from projects where license = :#lic and id > :#min order by id")
```

Though if the message body is a `java.util.Map` then the named parameters will be taken from the body.

```
from("direct:projects")
  .to("sql:select * from projects where license = :#lic and id > :#min order by id")
```

3.81.10. Using expression parameters

Available as of Camel 2.14

In the given route below, we want to get all the project from the database. It uses the body of the exchange for defining the license and uses the value of a property as the second parameter.

```
from("direct:projects")
  .setBody(constant("ASF"))
  .setProperty("min", constant(123))
  .to("sql:select * from projects where license = :#{body} and id > :#{property.min}
order by id")
```

3.81.11. Using the JDBC based idempotent repository

Available as of Camel 2.7: In this section we will use the JDBC based idempotent repository.

From Camel 2.9 onwards there is an abstract class `org.apache.camel.processor.idempotent.jdbc.AbstractJdbcMessageIdRepository` you can extend to build custom JDBC idempotent repository.

First we have to create the database table which will be used by the idempotent repository. For **Camel 2.7**, we use the following schema:

```
CREATE TABLE CAMEL_MESSAGEPROCESSED (
  processorName VARCHAR(255),
  messageId VARCHAR(100)
)
```

In **Camel 2.8**, we added the `createdAt` column:

```
CREATE TABLE CAMEL_MESSAGEPROCESSED (
  processorName VARCHAR(255),
  messageId VARCHAR(100),
  createdAt TIMESTAMP
)
```

The SQL Server **TIMESTAMP** type is a fixed-length binary-string type. It does not map to any of the JDBC time types: **DATE**, **TIME**, or **TIMESTAMP**.

We recommend to have a unique constraint on the columns processorName and messageId. Because the syntax for this constraint differs for database to database, we do not show it here.

Second we need to setup a `javax.sql.DataSource` in the spring XML file:

```
<jdbc:embedded-database id="dataSource" type="DERBY" />
```

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>
```

3.81.12. Using the JDBC based aggregation repository



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 <code>javax.sql.DataSource</code> to use for accessing the database.
repositoryName	String	Mandatory: The name of the repository.
transactionManager	TransactionManager	Mandatory: The <code>org.springframework.transaction.PlatformTransactionManager</code> to manage transactions for the database. The <code>TransactionManager</code> must be able to support databases.
lobHandler	LobHandler	A <code>org.springframework.jdbc.support.lob.LobHandler</code> to handle Lob types in the database. Use this option to use a vendor specific <code>LobHandler</code> , 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 <code>false</code> to optimize as we do not need the old exchange when aggregating.

Option	Type	Description
<code>useRecovery</code>	boolean	Whether or not recovery is enabled. This option is by default <code>true</code> . When enabled the Camel Aggregator automatic recover failed aggregated exchange and have them resubmitted.
<code>recoveryInterval</code>	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.
<code>maximumRedeliveries</code>	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 <code>deadLetterUri</code> option must also be provided.
<code>deadLetterUri</code>	String	An endpoint uri for a Dead Letter Channel where exhausted recovered Exchanges will be moved. If this option is used then the <code>maximumRedeliveries</code> option must also be provided.
<code>storeBodyAsText</code>	boolean	Starting with Camel 2.11, whether to store the message body as String which is human readable. By default this option is <code>false</code> meaning it is stored in binary format.
<code>headersToStoreAsText</code>	List <String>	Starting with Camel 2.11, allows for storing headers as a human-readable String. By default this option is disabled, meaning they will be stored in binary format.
<code>optimisticLocking</code>	false	Starting with Camel 2.12, to turn on optimistic locking, which often would be needed in clustered environments where multiple Camel applications shared the same JDBC based aggregation repository.
<code>jdbcOptimisticLockingExceptionMapper</code>		Starting with Camel 2.12, allows to plugin a custom <code>org.apache.camel.processor.aggregate.jdbc.JdbcOptimisticLockingExceptionMapper</code> to map vendor specific error codes to an optimistic locking error, for Camel to perform a retry. This requires <code>optimisticLocking</code> to be enabled.

3.81.12.1. What is preserved when persisting

`JdbcAggregationRepository` will preserve only `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.

Note from Camel 2.11 onwards you can store the message body and select(ed) headers as `String` in separate columns.

3.81.12.2. 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](#).

3.81.12.3. 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)
);
```

3.81.12.4. Storing body and headers as text

Available as of Camel 2.11

You can configure the `JdbcAggregationRepository` to store message body and select(ed) headers as String in separate columns. For example to store the body, and the following two headers `companyName` and `accountName` use the following SQL:

```
CREATE TABLE aggregationRepo3 (
  id varchar(255) NOT NULL,
  exchange blob NOT NULL,
  body varchar(1000),
  companyName varchar(1000),
  accountName varchar(1000),
  constraint aggregationRepo3_pk PRIMARY KEY (id)
);
```

```
CREATE TABLE aggregationRepo3_completed (  
    id varchar(255) NOT NULL,  
    exchange blob NOT NULL,  
    body varchar(1000),  
    companyName varchar(1000),  
    accountName varchar(1000),  
    constraint aggregationRepo3_completed_pk PRIMARY KEY (id)  
);
```

And then configure the repository to enable this behavior as shown below:

```
<bean id="repo3"  
    class="org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository">  
    <property name="repositoryName" value="aggregationRepo3"/>  
    <property name="transactionManager" ref="txManager3"/>  
    <property name="dataSource" ref="dataSource3"/>  
    <!-- configure to store the message body and following headers as text in the repo  
    -->  
    <property name="storeBodyAsText" value="true"/>  
    <property name="headersToStoreAsText">  
        <list>  
            <value>companyName</value>  
            <value>accountName</value>  
        </list>  
    </property>  
</bean>
```

3.81.12.5. 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.

3.81.12.6. Transaction

A Spring `PlatformTransactionManager` is required to orchestrate transaction.

3.81.12.7. 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.

3.81.12.8. 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>
```

3.81.12.9. Optimistic locking

From **Camel 2.12** onwards you can turn on `optimisticLocking` and use this JDBC based aggregation repository in a clustered environment where multiple Camel applications shared the same database for the aggregation repository. If there is a race condition there JDBC driver will throw a vendor specific exception which the `JdbcAggregationRepository` can react upon. To know which caused exceptions from the JDBC driver is regarded as an optimistic locking error we need a mapper to do this. Therefore there is a `org.apache.camel.processor.aggregate.jdbc.JdbcOptimisticLockingExceptionMapper` allows you to implement your custom logic if needed. There is a default implementation `org.apache.camel.processor.aggregate.jdbc.DefaultJdbcOptimisticLocking`

`ExceptionMapper` which works as follows:

The following check is done:

- If the caused exception is an `SQLException` then the `SQLState` is checked if starts with 23.
- If the caused exception is a `DataIntegrityViolationException`
- If the caused exception class name has "ConstraintViolation" in its name.
- optional checking for FQN class name matches if any class names has been configured

You can in addition add FQN classnames, and if any of the caused exception (or any nested) equals any of the FQN class names, then its an optimistic locking error.

Here is an example, where we define 2 extra FQN class names from the JDBC vendor.

```
<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"/>
    <property name="jdbcOptimisticLockingExceptionMapper" ref="myExceptionMapper"/>
</bean>

<!-- use the default mapper with extra FQN class names from our JDBC driver -->
```

```
<bean id="myExceptionHandler"
  class="org.apache.camel.processor.aggregate.jdbc.DefaultJdbcOptimisticLocking
  ExceptionMapper">
  <property name="classNames">
    <util:set>
      <value>com.foo.sql.MyViolationExceptoion</value>
      <value>com.foo.sql.MyOtherViolationExceptoion</value>
    </util:set>
  </property>
</bean>
```

3.82. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.82.1. URI Format

ssh:[username[:password]@]host[:port][?options]

3.82.2. 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 before 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.

Name	Default Value	Description
pollCommand		Consumer only: Command to send to SSH Server during each poll cycle. Used only when acting as Consumer.

3.83. StAX

Available as of Camel 2.9

The StAX component allows messages to be process through a SAX [ContentHandler](#).

Another feature of this component is to allow to iterate over JAXB records using StAX, for example using the [Splitter](#) EIP.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-stax</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.83.1. URI format

```
stax:content-handler-class
```

example:

```
stax:org.superbiz.FooContentHandler
```

From **Camel 2.11.1** onwards you can lookup a `org.xml.sax.ContentHandler` bean from the [Registry](#) using the `#` syntax as shown:

```
stax:#myHandler
```

3.83.2. Usage of a content handler as StAX parser

The message body after the handling is the handler itself.

Here an example:

```
from("file:target/in")
  .to("stax:org.superbiz.handler.CountingHandler")
  // CountingHandler implements org.xml.sax.ContentHandler or extends org.xml.sax.
  helpers.DefaultHandler
  .process(new Processor() {
    @Override
    public void process(Exchange exchange) throws Exception {
      CountingHandler handler = exchange.getIn().getBody(CountingHandler.class);
      // do some great work with the handler
    }
  });
```

3.83.3. Iterate over a collection using JAXB and StAX

First we suppose you have JAXB objects.

For instance a list of records in a wrapper object:

```
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "records")
public class Records {
    @XmlElement(required = true)
    protected List<Record> record;

    public List<Record> getRecord() {
        if (record == null) {
            record = new ArrayList<Record>();
        }
        return record;
    }
}
```

and

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "record", propOrder = { "key", "value" })
public class Record {
    @XmlAttribute(required = true)
    protected String key;

    @XmlAttribute(required = true)
    protected String value;

    public String getKey() {
        return key;
    }

    public void setKey(String key) {
        this.key = key;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

Then you get a XML file to process:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<records>
  <record value="v0" key="0"/>
  <record value="v1" key="1"/>
</records>
```



```
<record value="v2" key="2"/>
<record value="v3" key="3"/>
<record value="v4" key="4"/>
<record value="v5" key="5"/>
</record>
```

The StAX component provides an `StAXBuilder` which can be used when iterating XML elements with the Camel [Splitter](#)

```
from("file:target/in")
    .split(stax(Record.class)).streaming()
    .to("mock:records");
```

Where `stax` is a static method on `org.apache.camel.component.stax.StAXBuilder` which you can static import in the Java code. The stax builder is by default namespace aware on the `XMLReader` it uses. From **Camel 2.11.1** onwards you can turn this off by setting the boolean parameter to false, as shown below:

```
from("file:target/in")
    .split(stax(Record.class, false)).streaming()
    .to("mock:records");
```

3.83.3.1. The previous example with XML DSL

The example above could be implemented as follows in XML DSL

```
<!-- use STaXBuilder to create the expression we want to use in the route below for
splitting the XML file -->
<!-- notice we use the factory-method to define the stax method, and to pass in the
parameter as a constructor-arg -->
<bean id="staxRecord" class="org.apache.camel.component.stax.StAXBuilder" factory-
method="stax">
    <!-- FQN class name of the POJO with the JAXB annotations -->
    <constructor-arg index="0" value="org.apache.camel.component.stax.model.Record"/>
</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <!-- pickup XML files -->
        <from uri="file:target/in"/>
        <split streaming="true">
            <!-- split the file using StAX (ref to bean above) -->
            <!-- and use streaming mode in the splitter -->
            <ref>staxRecord</ref>
            <!-- and send each splitted to a mock endpoint, which will be a Record POJO
instance -->
            <to uri="mock:records"/>
        </split>
    </route>
</camelContext>>0
```

3.84. Stomp

Available as of Camel 2.12

The **stomp** component is used for communicating with [Stomp](#) compliant message brokers, like [Apache ActiveMQ](#) or [ActiveMQ Apollo](#)

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-stomp</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.84.1. URI format

```
stomp:queue:destination[?options]
```

Where **destination** is the name of the queue.

3.84.2. Options

Property	Default	Description
brokerURL	tcp://localhost:61613	The URI of the Stomp broker to connect to
login		The username
passcode		The password

You can append query options to the URI in the following format, ?option=value&option=value&...

3.84.3. Samples

Sending messages:

```
from("direct:foo").to("stomp:queue:test");
```

Consuming messages:

```
from("stomp:queue:test").transform(body().convertToString()).to("mock:result")
```

3.84.4. Endpoints

Camel supports the [Message Endpoint](#) pattern using the [Endpoint](#) interface. Endpoints are usually created by a [Component](#) and Endpoints are usually referred to in the [DSL](#) via their [URIs](#).

From an Endpoint you can use the following methods

- [createProducer\(\)](#) will create a [Producer](#) for sending message exchanges to the endpoint
- [createConsumer\(\)](#) implements the [Event Driven Consumer](#) pattern for consuming message exchanges from the endpoint via a [Processor](#) when creating a [Consumer](#)
- [createPollingConsumer\(\)](#) implements the [Polling Consumer](#) pattern for consuming message exchanges from the endpoint via a [PollingConsumer](#)

3.85. 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.

3.85.1. URI Format

```
stub:someUri
```

Where `someUri` can be any URI with any query parameters.

3.85.2. Samples

Here are some samples:

```
stub:smtp://somehost.foo.com?user=whatnot&something=else
```

```
stub:http://somehost.bar.com/something
```

3.86. Test

[Testing](#) of distributed and asynchronous processing is notoriously difficult. The [Mock](#), [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 [Mock](#) component to support pulling messages from another endpoint on startup to set the expected message bodies on the underlying [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 [Mock](#) endpoint, which is only valid if the received messages match the number of expected messages and their message payloads are equal.

3.86.1. URI format

```
test:expectedMessagesEndpointUri
```

where **expectedMessagesEndpointUri** refers to some other Component URI that the expected message bodies are pulled from before starting the test.

3.86.2. URI Options

Name	Default Value	Description
timeout	2000	Camel 2.12: The timeout to use when polling for message bodies from the URI.

3.86.3. 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.

To see how you can set other expectations on the test endpoint, see [Mock](#) component.

3.87. Timer

The **timer:** component is used to generate message exchanges when a timer fires. You can only consume events from this endpoint.

3.87.1. 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`.



See also the [Quartz](#) component that supports much more advanced scheduling.



You can specify the time in [human friendly syntax](#).

3.87.2. Options

Name	Default Value	Description
time	null	A <code>java.util.Date</code> the first event should be generated. If using the URI, the pattern expected is: <code>yyyy-MM-dd HH:mm:ss</code> or <code>yyyy-MM-dd'T'HH:mm:ss</code> .

Name	Default Value	Description
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 / 1000	The number of milliseconds to wait before the first event is generated. Should not be used in conjunction with the time option. Starting with Camel 2.11 the default value is 1000, versions prior to that 0.
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.

3.87.3. 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.

3.87.4. 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

3.87.5. 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&period=60000"/>
  <to uri="bean:myBean?method=someMethodName"/>
</route>
```

3.88. Twitter

Available as of Camel 2.10

The Twitter component enables the most useful features of the Twitter API by encapsulating [Twitter4J](#). It allows direct, polling, or event-driven consumption of timelines, users, trends, and direct messages. Also, it supports producing messages as status updates or direct messages.

Twitter now requires the use of OAuth for all client application authentication. In order to use camel-twitter with your account, you'll need to create a new application within Twitter at <https://dev.twitter.com/apps/new> and grant the application access to your account. Finally, generate your access token and secret.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-twitter</artifactId>
  <version>${camel-version}</version>
</dependency>
```

3.88.1. URI format

```
twitter://endpoint[?options]
```

3.88.2. TwitterComponent:

The twitter component can be configured with the Twitter account settings which is mandatory to configure before using.

You can also configure these options directly in the endpoint.

Option	Description
consumerKey	The consumer key
consumerSecret	The consumer secret
accessToken	The access token
accessTokenSecret	The access token secret

3.88.3. Consumer Endpoints:

Rather than the endpoints returning a List through one single route exchange, camel-twitter creates one route exchange per returned object. As an example, if "timeline/home" results in five statuses, the route will be executed five times (one for each Status).

Endpoint	Context	Body Type	Notice
directmessage	direct, polling	twitter4j.DirectMessage	
search	direct, polling	twitter4j.Tweet	
streaming/filter	event, polling	twitter4j.Status	
streaming/sample	event, polling	twitter4j.Status	
timeline/home	direct, polling	twitter4j.Status	

Endpoint	Context	Body Type	Notice
timeline/mentions	direct, polling	twitter4j.Status	
timeline/public	direct, polling	twitter4j.Status	@deprecated. Use timeline/home or direct/home instead. Removed from Camel 2.11 onwards.
timeline/retweetsofme	direct, polling	twitter4j.Status	
timeline/user	direct, polling	twitter4j.Status	
trends/daily	Camel 2.10.1: direct, polling	twitter4j.Status	@deprecated. Removed from Camel 2.11 onwards.
trends/weekly	Camel 2.10.1: direct, polling	twitter4j.Status	@deprecated. Removed from Camel 2.11 onwards.

3.88.4. Producer Endpoints:

Endpoint	Body Type
directmessage	String
search	List<twitter4j.Tweet>
timeline/user	String

3.88.5. URI Options

Name	Default Value	Description
type	direct	direct, event, or polling
delay	60	in seconds
consumerKey	null	Consumer Key. Can also be configured on the <code>TwitterComponent</code> level instead.
consumerSecret	null	Consumer Secret. Can also be configured on the <code>TwitterComponent</code> level instead.
accessToken	null	Access Token. Can also be configured on the <code>TwitterComponent</code> level instead.
accessTokenSecret	null	Access Token Secret. Can also be configured on the <code>TwitterComponent</code> level instead.
user	null	Username, used for user timeline consumption, direct message production, etc.
locations	null	'lat,lon;lat,lon;...' Bounding boxes, created by pairs of lat/lons. Can be used for streaming/filter
keywords	null	'foo1,foo2,foo3...' Can be used for search and streaming/filter. See Advanced search for keywords syntax for searching with for example OR.
userIds	null	'username,username...' Can be used for streaming/filter
filterOld	true	Filter out old tweets, that has previously been polled. This state is stored in memory only, and based on last tweet id. Since Camel 2.11.0 The search producer supports this option
sinceId	1	Camel 2.11.0: The last tweet id which will be used for pulling the tweets. It is useful when the camel route is restarted after a long running.
lang	null	Camel 2.11.0: The lang string ISO_639-1 which will be used for searching
count	null	Camel 2.11.0: Limiting number of results per page.
numberOfPages	1	Camel 2.11.0: The number of pages result which you want camel-twitter to consume.
httpProxyHost	null	Camel 2.12.3: The http proxy host which can be used for the camel-twitter.
httpProxyPort	null	Camel 2.12.3: The http proxy port which can be used for the camel-twitter.
httpProxyUser	null	Camel 2.12.3: The http proxy user which can be used for the camel-twitter.
httpProxyPassword	null	Camel 2.12.3: The http proxy password which can be used for the camel-twitter.
useSSL	true	Camel 2.12.3: Using the SSL to connect the api.twitter.com if the option is true.

3.88.6. Message header

Name	Description
CamelTwitterKeywords	This header is used by the search producer to change the search key words dynamically.
CamelTwitterSearchLanguage	Camel 2.11.0: This header can override the option of <code>lang</code> which set the search language for the search endpoint dynamically
CamelTwitterCount	Camel 2.11.0 This header can override the option of <code>count</code> which sets the max twitters that will be returned.
CamelTwitterNumberOfPages	Camel 2.11.0 This header can convert the option of <code>numberOfPages</code> which sets how many pages we want to twitter returns.

3.88.7. Message body

All message bodies utilize objects provided by the Twitter4J API.

3.88.8. Use cases

3.88.8.1. To create a status update within your Twitter profile, send this producer a String body.

```
from("direct:foo")
  .to("twitter://timeline/user?consumerKey=[s]&consumerSecret=[s]&accessToken=[s]&accessTokenSecret=[s]");
```

3.88.8.2. To poll, every 5 sec., all statuses on your home timeline:

```
from("twitter://timeline/home?type=polling&delay=5&consumerKey=[s]&consumerSecret=[s]&accessToken=[s]&accessTokenSecret=[s]")
  .to("bean:blah");
```

3.88.8.3. To search for all statuses with the keyword 'camel':

```
from("twitter://search?type=direct&keywords=camel&consumerKey=[s]&consumerSecret=[s]&accessToken=[s]&accessTokenSecret=[s]")
  .to("bean:blah");
```

3.88.8.4. Searching using a producer with static keywords

```
from("direct:foo")
  .to("twitter://search?keywords=camel&consumerKey=[s]&consumerSecret=[s]&accessToken=[s]&accessTokenSecret=[s]");
```


3.88.8.5. Searching using a producer with dynamic keywords from header

In the bar header we have the keywords we want to search, so we can assign this value to the CamelTwitterKeywords header.

```
from("direct:foo")
  .setHeader("CamelTwitterKeywords", header("bar"))
  .to("twitter://search?consumerKey=[s]&consumerSecret=[s]&accessToken=[s]&access
TokenSecret=[s]");
```

3.88.9. Example

See also the [Twitter Websocket Example](#).

3.89. 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>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.89.1. 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&...`

3.89.2. Options

Option	Default	Description
loaderCache	true	Velocity based file loader cache.
contentCache	true	Cache for the resource content when it is loaded. Note: as of Camel 2.9 cached resource content can be cleared via JMX using the endpoint's <code>clearContentCache</code> operation.
encoding	null	Character encoding of the resource content.
propertiesFile	null	The URI of the properties file which is used for VelocityEngine initialization.

3.89.3. 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
CamelVelocityResourceUri	The templateName as a <code>String</code> 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 `.vm`:

```
$in.setHeader('fruit', 'Apple')
```

The `fruit` header is now accessible from the `message.out.headers`.

3.89.4. Velocity Context

Camel will provide exchange information in the Velocity context (just a `Map`). The `Exchange` is transferred as:

key	value
exchange	The <code>Exchange</code> itself.
headers	The headers of the In message.
camelContext	The Camel Context instance.
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).

3.89.5. 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.

3.89.6. 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.

3.89.7. 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" );
```

3.90. Vertx

Available as of Camel 2.12

The **vertx** component is for working with the [VertxEventBus](#).

The vertx [EventBus](#) sends and receives JSON events.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-vertx</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
```

```
</dependency>
```

3.90.1. URI format

```
vertex:channelName[?options]
```

3.90.2. Options

Name	Default Value	Description
pubSub	false	Camel 2.12.3: Whether to use publish/subscribe instead of point to point when sending to a vertex endpoint.

You can append query options to the URI in the following format, ?option=value&option=value&...

3.91. VM

The **vm:** component provides asynchronous [SEDA](#) behavior, exchanging messages on a [BlockingQueue](#) and invoking consumers in a separate thread pool.

This component differs from the [SEDA](#) component in that VM supports communication across CamelContext instances so you can use this mechanism to communicate across web applications (provided that camel-core.jar is on the system/boot classpath).

VM is an extension to the [SEDA](#) component.

3.91.1. URI format

```
vm:queueName[?options]
```

where **queueName** can be any string to uniquely identify the endpoint within the JVM (or at least within the classloader that loaded camel-core.jar)

You can append query options to the URI in the following format: ?option=value&option=value&...

Before Camel 2.3 - Same URI must be used for both producer and consumer

An exactly identical **VM** endpoint URI **must** be used for both the producer and the consumer endpoint. Otherwise, Camel will create a second **VM** endpoint despite that the queueName portion of the URI is identical. For example:

```
from("direct:foo").to("vm:bar?concurrentConsumers=5");  
from("vm:bar?concurrentConsumers=5").to("file://output");
```

Notice that we have to use the full URI, including options in both the producer and consumer.

In Camel 2.4 this has been fixed so that only the queue name must match. Using the queue name bar, we could rewrite the previous example as follows:

```
from("direct:foo").to("vm:bar");  
from("vm:bar?concurrentConsumers=5").to("file://output");
```

3.91.2. Options

See [SEDA](#) component for options and other important usage details as the same rules apply to the [VM](#) component.

3.91.3. Samples

In the route below we send exchanges across CamelContext instances to a VM queue named order.email:

```
from("direct:in").bean(MyOrderBean.class).to("vm:order.email");
```

And then we receive exchanges in some other Camel context (such as deployed in another .war application):

```
from("vm:order.email").bean(MyOrderEmailSender.class);
```

3.92. Weather

Available as of Camel 2.12

The **weather:** component is used for polling weather information from [Open Weather Map](#) - a site that provides free global weather and forecast information. The information is returned as a json String object.

Camel will poll for updates to the current weather and forecasts once per hour by default.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-weather</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

3.92.1. URI format

```
weather://<unused name>[?options]
```

3.92.2. Options

Property	Default	Description
location	null	If null Camel will try and determine your current location using the geolocation of your ip address, else specify the city,country. For well known city names, Open Weather Map will determine the best fit, but multiple results may be returned. Hence specifying and country as well will return more accurate data. If you specify "current" as the location then the component will try to get the current latitude and longitude and use that to get the weather details. You can use lat and lon options instead of location.
lat	null	Latitude of location. You can use lat and lon options instead of location.

Property	Default	Description
lon	null	Longitude of location. You can use lat and lon options instead of location.
period	null	If null, the current weather will be returned, else use values of 5, 7, 14 days. Only the numeric value for the forecast period is actually parsed, so spelling, capitalisation of the time period is up to you (its ignored)
headerName	null	To store the weather result in this header instead of the message body. This is useable if you want to keep current message body as-is.
mode	JSON	The output format of the weather data. The possible values are HTML, JSON or XML
units	METRIC	The units for temperature measurement. The possible values are IMPERIAL or METRIC
consumer.delay	3600000	Delay in millis between each poll (default is 1 hour)
consumer.initialDelay	1000	Millis before polling starts.
consumer.userFixedDelay	false	If true, use fixed delay between polls, 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&...`

3.92.3. Exchange data format

Camel will deliver the body as a json formatted `java.lang.String` (see the `mode` option above).

3.92.4. Message Headers

Header	Description
CamelWeatherQuery	The original query URL sent to the Open Weather Map site
CamelWeatherLocation	Used by the producer to override the endpoint location and use the location from this header instead.

3.92.5. Samples

In this sample we find the 7 day weather forecast for Madrid, Spain:

```
from("weather:foo?location=Madrid,Spain&period=7 days").to("jms:queue:weather");
```

To just find the current weather for your current location you can use this:

```
from("weather:foo").to("jms:queue:weather");
```

And to find the weather using the producer we do:

```
from("direct:start").to("weather:foo?location=Madrid,Spain");
```

And we can send in a message with a header to get the weather for any location as shown:

```
String json = template.requestBodyAndHeader("direct:start", "",
"CamelWeatherLocation", "Paris,France", String.class);
```

And to get the weather at the current location, then:

```
String json = template.requestBodyAndHeader("direct:start", "",
"CamelWeatherLocation", "current", String.class);
```

3.93. Websocket

Available as of Camel 2.10

The **websocket** component provides websocket [endpoints](#) for communicating with clients using websocket. The component uses Eclipse Jetty Server which implements the [IETF](#) specification (drafts and RFC 6455). It supports the protocols ws:// and wss://. To use wss:// protocol, the SSLContextParameters must be defined.

Version currently supported:

As Camel 2.10 uses Jetty 7.5.4.v20111024, only the D00 to [D13](#) IETF implementations are available.

Camel 2.11 uses Jetty 7.6.7.

3.93.1. URI format

```
websocket://hostname[:port][/resourceUri][?options]
```

You can append query options to the URI in the following format, ?option=value&option=value&...

3.93.2. Component Options

The WebsocketComponent can be configured prior to use, to setup host, to act as a websocket server.

Option	Default	Description
host	0.0.0.0	The hostname.
port	9292	The port number.
staticResources	null	Path for static resources such as index.html files etc. If this option has been configured, then a server is started on the given hostname and port, to service the static resources, eg such as an index.html file. If this option has not been configured, then no server is started.
sslContextParameters		Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry . This reference overrides any configured SSLContextParameters at the component level. See Using the JSSE Configuration Utility .
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.

3.93.3. Endpoint Options

The WebsocketEndpoint can be configured prior to use

Option	Default	Description
sslContextParametersRef		Deprecated and will be removed in Camel 3.0: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry . This

Option	Default	Description
		reference overrides any configured SSLContextParameters at the component level. See Using the JSSE Configuration Utility . Use the sslContextParameters option instead
sslContextParameters		Camel 2.11.1: Reference to a <code>org.apache.camel.util.jsse.SSLContextParameters</code> in the Registry . This reference overrides any configured SSLContextParameters at the component level. See Using the JSSE Configuration Utility .
sendToAll	null	Producer only: To send to all websocket subscribers. Can be used to configure on endpoint level, instead of having to use the <code>WebsocketConstants.SEND_TO_ALL</code> header on the message.
staticResources	null	The root directory for the web resources or classpath. Use the protocol file: or classpath: depending if you want that the component loads the resource from file system or classpath.
bufferSize	null	Camel 2.12.3: set the buffer size of the websocketServlet, which is also the max frame byte size (default 8192)
maxIdleTime	null	Camel 2.12.3: set the time in ms that the websocket created by the websocketServlet may be idle before closing. (default is 300000)
maxTextMessageSize	null	Camel 2.12.3: can be used to set the size in characters that the websocket created by the websocketServlet may be accept before closing.
maxBinaryMessageSize	null	Camel 2.12.3: can be used to set the size in bytes that the websocket created by the websocketServlet may be accept before closing. (Default is -1 - or unlimited)
minVersion	null	Camel 2.12.3: can be used to set the minimum protocol version accepted for the websocketServlet. (Default 13 - the RFC6455 version)

3.93.4. Message Headers

The websocket component uses 2 headers to indicate to either send messages back to a single/current client, or to all clients.

Key	Description
<code>WebsocketConstants.SEND_TO_ALL</code>	Sends the message to all clients which are currently connected. You can use the <code>sendToAll</code> option on the endpoint instead of using this header.
<code>WebsocketConstants.CONNECTION_KEY</code>	Sends the message to the client with the given connection key.

3.93.5. Usage

In this example we let Camel exposes a websocket server which clients can communicate with. The websocket server uses the default host and port, which would be `0.0.0.0:9292`.

The example will send back an echo of the input. To send back a message, we need to send the transformed message to the same endpoint `"websocket://echo"`. This is needed

because by default the messaging is InOnly.

```
// expose a echo websocket client, that sends back an echo
from("websocket://echo")
    .log(">>> Message received from WebSocket Client : ${body}")
    .transform().simple("${body}${body}")
    // send back to the client, by sending the message to the same endpoint
```



```
// this is needed as by default messages is InOnly
// and we will by default send back to the current client using the provided
connection key
.to("websocket://echo");
```

This example is part of an unit test, which you can find [here](#). As a client we use the [AHC](#) library which offers support for web socket as well.

Here is another example where webapp resources location have been defined to allow the Jetty Application Server to not only register the WebSocket servlet but also to expose web resources for the browser. Resources should be defined under the webapp directory.

```
from("activemq:topic:newsTopic")
    .routeId("fromJMStoWebSocket")
    .to("websocket://localhost:8443/newsTopic?sendToAll=true&staticResources=
classpath:webapp");
```

3.93.6. Setting up SSL for WebSocket Component

3.93.6.1. Using the JSSE Configuration Utility

As of Camel 2.10, the WebSocket 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 Cometd 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");

TrustManagersParameters tmp = new TrustManagersParameters();
tmp.setKeyStore(ksp);

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);
scp.setTrustManagers(tmp);

CometdComponent cometdComponent = getContext().getComponent("cometds",
CometdComponent.class);
cometdComponent.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:trustManagers>
            <camel:keyStore
                resource="/users/home/server/keystore.jks"
                password="keystorePassword"/>
            </camel:trustManagers>
        </camel:sslContextParameters>...
    ...
    <to uri="websocket://127.0.0.1:8443/test?sslContextParameters=#sslContextParameters"/>...

```

Java DSL based configuration of endpoint

```

...
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() {

            String uri = "websocket://127.0.0.1:8443/test?sslContextParameters=#sslContextParameters";

            from(uri)
                .log(">>> Message received from WebSocket Client : ${body}")
                .to("mock:client")
                .loop(10)
                .setBody().constant(">>> Welcome on board!")
                .to(uri);
        }
    };
}
...

```

3.94. 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>

```

3.94.1. URI format

```
xquery:templateName[?options]
```

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");
```

3.95. 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.

3.95.1. 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.xml	refers to the file com/acme/mytransform.xml on the classpath
xslt:file:///foo/bar.xml	refers to the file /foo/bar.xml
xslt:http://acme.com/cheese/foo.xml	refers to the remote http resource

Maven users will need to add the following dependency to their `pom.xml` for this component when using **Camel 2.8** or older:

```
<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>
```

From Camel 2.9 onwards the [XSLT](#) component is provided directly in the camel-core.

3.95.2. 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 <code>org.apache.camel.converter.jaxp.XmlConverter</code> .

Name	Default Value	Description
transformerFactory	null	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	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.transform.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.
deleteOutputFile	false	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	Cache for the resource content (the stylesheet file) when it is loaded. If set to false Camel will reload the stylesheet file on each message processing. A cached stylesheet can be forced to reload at runtime via JMX using the clearCachedStylesheet operation.
allowStAX	false	Whether to allow using StAX as the javax.xml.transform.Source.
transformerCacheSize	0	The number of javax.xml.transform.Transformer objects that are cached for reuse to avoid calls to Template.newTransformer().
saxon	false	Available in Camel 2.11. Whether to use Saxon as the transformerFactoryClass. If enabled then the class net.sf.saxon.TransformerFactoryImpl will be used. Saxon will need to be available in the classpath in this case.

3.95.3. 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");
```

3.95.4. 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 usable.

```
<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 ...>
```

3.95.5. 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.

3.95.6. 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.xml"/>
```

Which then will resolve the xsl file under `org/apache/camel/component`.

3.95.7. Using xsl:include and default prefix

When using xsl:include such as:

```
<xsl:include href="staff_template.xml"/>
```

Then in Camel 2.10.3 and older, then Camel will use "classpath:" as the default prefix, and load the resource from the classpath. This works for most cases, but if you configure the starting resource to load from file,

```
.to("xslt:file:etc/xslt/staff_include_relative.xml")
```

.. then you would have to prefix all your includes with "file:" as well.

```
<xsl:include href="file:staff_template.xml"/>
```

From Camel 2.10.4 onwards we have made this easier as Camel will use the prefix from the endpoint configuration as the default prefix. So from Camel 2.10.4 onwards you can do:

```
<xsl:include href="staff_template.xml"/>
```

Which will load the `staff_template.xml` resource from the file system, as the endpoint was configured with "file:" as prefix.

You can still though explicit configure a prefix, and then mix and match. And have both file and classpath loading. But that would be unusual, as most people either use file or classpath based resources.

3.96. Yammer

Available as of Camel 2.12

The Yammer component allows you to interact with the [Yammer](#) enterprise social network. Consuming messages, users, and user relationships is supported as well as creating new messages.

Yammer uses OAuth 2 for all client application authentication. In order to use camel-yammer with your account, you'll need to create a new application within Yammer and grant the application access to your account. Finally, generate your access token. More details are at <https://developer.yammer.com/authentication/>

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-yammer</artifactId>
  <version>${camel-version}</version>
</dependency>
```

3.96.1. URI format

```
yammer:[function]?[options]
```

3.96.2. YammerComponent

The yammer component can be configured with the Yammer account settings which are mandatory to configure before using. You can also configure these options directly in the endpoint.

Option	Description
consumerKey	The consumer key
consumerSecret	The consumer secret
accessToken	The access token

3.96.3. Consuming messages

The camel-yammer component provides several endpoints for consuming messages:

URI	Description
yammer:messages?[options]	All public messages in the user's (whose access token is being used to make the API call) Yammer network. Corresponds to "All" conversations in the Yammer web interface.
yammer:my_feed?[options]	The user's feed, based on the selection they have made between "Following" and "Top" conversations.
yammer:algo?[options]	The algorithmic feed for the user that corresponds to "Top" conversations, which is what the vast majority of users will see in the Yammer web interface.
yammer:following?[options]	The "Following" feed which is conversations involving people, groups and topics that the user is following.
yammer:sent?[options]	All messages sent by the user.
yammer:private?[options]	Private messages received by the user.
yammer:received?[options]	Camel 2.12.1: All messages received by the user

3.96.3.1. URI Options for consuming messages

Name	Default Value	Description
useJson	false	Set to true if you want to use raw JSON rather than converting to POJOs.
delay	5000	in milliseconds
consumerKey	null	Consumer Key. Can also be configured on the YammerComponent level instead.
consumerSecret	null	Consumer Secret. Can also be configured on the YammerComponent level instead.
accessToken	null	Access Token. Can also be configured on the YammerComponent level instead.
limit	-1	Return only the specified number of messages. Works for threaded=true and threaded=extended.
threaded	null	threaded=true will only return the first message in each thread. This parameter is intended for apps which display message threads collapsed. threaded=extended will return the thread starter messages in order of most recently active as well as the two most recent messages, as they are viewed in the default view on the Yammer web interface.
olderThan	-1	Returns messages older than the message ID specified as a numeric string. This is useful for paginating messages. For example, if you're currently viewing 20 messages and the oldest is number 2912, you could append "?olderThan=2912#" to your request to get the 20 messages prior to those you're seeing.
newerThan	-1	Returns messages newer than the message ID specified as a numeric string. This should be used when polling for new messages. If you're looking at messages, and the most recent

Name	Default Value	Description
		message returned is 3516, you can make a request with the parameter "?newerThan=3516#" to ensure that you do not get duplicate copies of messages already on your page.

3.96.3.2. Message format

All messages by default are converted to a POJO model provided in the `org.apache.camel.component.yammer.model` package. The original message coming from yammer is in JSON. For all message consuming & producing endpoints, a Messages object is returned. Take for example a route like:

```
from("yammer:messages?consumerKey=aConsumerKey&consumerSecret=aConsumerSecretKey&accessToken=aAccessToken").to("mock:result");
```

and lets say the yammer server returns:

```
{
  "messages": [
    {
      "replied_to_id": null,
      "network_id": 7654,
      "url": "https://www.yammer.com/api/v1/messages/305298242",
      "thread_id": 305298242,
      "id": 305298242,
      "message_type": "update",
      "chat_client_sequence": null,
      "body": {
        "parsed": "Testing yammer API...",
        "plain": "Testing yammer API...",
        "rich": "Testing yammer API..."
      },
      "client_url": "https://www.yammer.com/",
      "content_excerpt": "Testing yammer API...",
      "created_at": "2013/06/25 18:14:45 +0000",
      "client_type": "Web",
      "privacy": "public",
      "sender_type": "user",
      "liked_by": {
        "count": 1,
        "names": [
          {
            "permalink": "janstey",
            "full_name": "Jonathan Anstey",
            "user_id": 1499642294
          }
        ]
      }
    },
    {
      "sender_id": 1499642294,
      "language": null,
      "system_message": false,
      "attachments": [
      ],
      "direct_message": false,
      "web_url": "https://www.yammer.com/redhat.com/messages/305298242"
    },
    {
      "replied_to_id": null,
      "network_id": 7654,
      "url": "https://www.yammer.com/api/v1/messages/294326302",
      "thread_id": 294326302,
      "id": 294326302,
```



```

    "message_type": "system",
    "chat_client_sequence": null,
    "body": {
      "parsed": "(Principal Software Engineer) has [[tag:14658]] the redhat.com network. Take a moment to welcome Jonathan.",
      "plain": "(Principal Software Engineer) has #joined the redhat.com network. Take a moment to welcome Jonathan.",
      "rich": "(Principal Software Engineer) has #joined the redhat.com network. Take a moment to welcome Jonathan."
    },
    "client_url": "https://www.yammer.com/",
    "content_excerpt": "(Principal Software Engineer) has #joined the redhat.com network. Take a moment to welcome Jonathan.",
    "created_at": "2013/05/10 19:08:29 +0000",
    "client_type": "Web",
    "sender_type": "user",
    "privacy": "public",
    "liked_by": {
      "count": 0,
      "names": [
        ]
      }
    }
  ]
}

```

Camel will marshal that into a Messages object containing 2 Message objects. As shown below there is a rich object model that makes it easy to get any information you need:

```

Exchange exchange = mock.getExchanges().get(0);
Messages messages = exchange.getIn().getBody(Messages.class);

assertEquals(2, messages.getMessage().size());
assertEquals("Testing yammer API...", messages.getMessage().get(0).getBody().getPlain());
assertEquals("(Principal Software Engineer) has #joined the redhat.com network. Take a moment to welcome Jonathan.",
  messages.getMessage().get(1).getBody().getPlain());

```

That said, marshaling this data into POJOs is not free so if you need you can switch back to using pure JSON by adding the `useJson=false` option to your URI.

3.96.4. Creating messages

To create a new message in the account of the current user, you can use the following URI:

```
yammer:messages?[options]
```

The current Camel message body is what will be used to set the text of the Yammer message. The response body will include the new message formatted the same way as when you consume messages (i.e. as a Messages object by default).

Take this route for instance:

```

from("direct:start").to("yammer:messages?consumerKey=aConsumerKey&consumerSecret=aConsumerSecretKey&accessToken=aAccessToken").to("mock:result");

```

By sending to the `direct:start` endpoint a "Hi from Camel!" message body:

```
template.sendBody("direct:start", "Hi from Camel!");
```

a new message will be created in the current user's account on the server and also this new message will be returned to Camel and converted into a Messages object. Like when consuming messages you can interrogate the Messages object:

```
Exchange exchange = mock.getExchanges().get(0);
Messages messages = exchange.getIn().getBody(Messages.class);

assertEquals(1, messages.getMessages().size());
assertEquals("Hi from Camel!", messages.getMessages().get(0).getBody().getPlain());
```

3.96.5. Retrieving user relationships

The camel-yammer component can retrieve user relationships:

```
yammer:relationships?[options]
```

3.96.5.1. URI Options for retrieving relationships

Name	Default Value	Description
useJson	false	Set to true if you want to use raw JSON rather than converting to POJOs.
delay	5000	in milliseconds
consumerKey	null	Consumer Key. Can also be configured on the <code>YammerComponent</code> level instead.
consumerSecret	null	Consumer Secret. Can also be configured on the <code>YammerComponent</code> level instead.
accessToken	null	Access Token. Can also be configured on the <code>YammerComponent</code> level instead.
userId	current user	To view the relationships for a user other than the current user.

3.96.6. Retrieving users

The camel-yammer component provides several endpoints for retrieving users:

URI	Description
yammer:users?[options]	Retrieve users in the current user's Yammer network.
yammer:current?[options]	View data about the current user.

3.96.6.1. URI Options for retrieving users

Name	Default Value	Description
useJson	false	Set to true if you want to use raw JSON rather than converting to POJOs.
delay	5000	in milliseconds
consumerKey	null	Consumer Key. Can also be configured on the <code>YammerComponent</code> level instead.
consumerSecret	null	Consumer Secret. Can also be configured on the <code>YammerComponent</code> level instead.
accessToken	null	Access Token. Can also be configured on the <code>YammerComponent</code> level instead.

3.96.7. Using an enricher

It is helpful sometimes (or maybe always in the case of users or relationship consumers) to use an enricher pattern rather than a route initiated with one of the polling consumers in camel-yammer. This is because the consumers will fire repeatedly, however often you set the delay for. If you just want to look up a user's data, or grab a message at a point in time, it is better to call that consumer once and then get one with your route.

Lets say you have a route that at some point needs to go out and fetch user data for the current user. Rather than polling for this user over and over again, use the pollEnrich DSL method:

```
from("direct:start").pollEnrich("yammer:current?consumerKey=aConsumerKey&consumerSecret=aConsumerSecretKey&accessToken=aAccessToken").to("mock:result");
```

This will go out and fetch the current user's User object and set it as the Camel message body.

3.97. Zookeeper

The ZooKeeper component allows interaction with a ZooKeeper cluster and exposes the following features to Camel:

- Creation of nodes in any of the ZooKeeper create modes.
- Get and Set the data contents of arbitrary cluster nodes.
- Create and retrieve the list the child nodes attached to a particular node.
- A Distributed RoutePolicy that leverages a Leader election coordinated by ZooKeeper to determine if exchanges should get processed.

Maven users will need to add the following dependency to their pom.xml for this component:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-zookeeper</artifactId>
  <!-- use the same version as your Camel core version -->
  <version>x.x.x</version>
</dependency>
```

3.97.1. URI Format and Options

```
zookeeper://zookeeper-server[:port][[/path]][?options]
```

The *path* from the uri specifies the node in the ZooKeeper server (aka znode) that will be the target of the endpoint.

Option can be:

Table 3.21.

Name	Default Value	Description
sessionId	null	The session id used to identify a connection to the cluster
password	null	The password to use when making a connection
listChildren	false	Whether the children of the node should be listed
repeat	false	Should changes to the znode be 'watched' and repeatedly processed.

Name	Default Value	Description
backoff	5000	The time interval to backoff for after an error before retrying.
timeout	5000	The time interval to wait on connection before timing out.
create	false	Should the endpoint create the node if it does not currently exist.
createMode	EPHEMERAL	The create mode that should be used for the newly created node (see below).
sendEmptyMessage-OnDelete	true	Upon the delete of a znode, should an empty message be sent to the consumer.

3.97.2. Use cases

3.97.2.1. Reading from a znode

The following snippet will read the data from the znode '/somepath/somenode/' provided that it already exists. The data retrieved will be placed into an exchange and passed onto the rest of the route.

```
from("zookeeper://localhost:39913/somepath/somenode").to("mock:result");
```

If the node does not yet exist then a flag can be supplied to have the endpoint await its creation:

```
from("zookeeper://localhost:39913/somepath/somenode?awaitCreation=true").
to("mock:result");
```

When data is read due to a WatchedEvent received from the ZooKeeper ensemble, the CamelZookeeperEventType header will hold the ZooKeeper's EventType value from that WatchedEvent. If the data is read initially (not triggered by a WatchedEvent) the CamelZookeeperEventType header will not be set.

3.97.2.2. Writing to a znode

The following snippet will write the payload of the exchange into the znode at '/somepath/somenode/' provided that it already exists:

```
from("direct:write-to-znode")
.to("zookeeper://localhost:39913/somepath/somenode");
```

For flexibility, the endpoint allows the target znode to be specified dynamically as a message header. If a header keyed by the string 'CamelZooKeeperNode' is present then the value of the header will be used as the path to the znode on the server. For instance using the same route definition above, the following code snippet will write the data not to '/somepath/somenode' but to the path from the header '/somepath/someothernode'

```
Exchange e = createExchangeWithBody(testPayload);
template.sendBodyAndHeader("direct:write-to-znode",
e, "CamelZooKeeperNode", "/somepath/someothernode");
```

To also create the node if it does not exist the 'create' option should be used.

```
from("direct:create-and-write-to-znode").
to("zookeeper://localhost:39913/somepath/somenode?create=true");
```

Starting with Camel 2.11 it will also be possible to delete a node using the header 'CamelZookeeperOperation' by setting it to 'DELETE':

```
from("direct:delete-znode")
    .setHeader(ZooKeeperMessage.ZOOKEEPER_OPERATION, constant("DELETE"))
    .to("zookeeper://localhost:39913/somepath/somenode");
```

Or equivalently:

```
<route>
  <from uri="direct:delete-znode" />
  <setHeader headerName="CamelZookeeperOperation">
    <constant>DELETE</constant>
  </setHeader>
  <to uri="zookeeper://localhost:39913/somepath/somenode" />
</route>
```

ZooKeeper nodes can have different types, they can be 'Ephemeral' or 'Persistent' and 'Sequenced' or 'Unsequenced'. Information of each type is described on the [ZooKeeper site](#). By default endpoints will create unsequenced, ephemeral nodes, but the type can be easily manipulated via a uri config parameter or via a special message header. The values expected for the create mode are simply the names from the CreateMode enumeration

- PERSISTENT
- PERSISTENT_SEQUENTIAL
- EPHEMERAL
- EPHEMERAL_SEQUENTIAL

For example to create a persistent znode via the URI config:

```
from("direct:create-and-write-to-persistent-znode")
    .to("zookeeper://localhost:39913/somepath/somenode?create=true //
&createMode=PERSISTENT");
```

Or using the header 'CamelZookeeperCreateMode'

```
Exchange e = createExchangeWithBody(testPayload);
template.sendBodyAndHeader("direct:create-and-write-to-persistent-znode",
    e, "CamelZookeeperCreateMode", "PERSISTENT");
```

3.97.3. ZooKeeper enabled Route policy

ZooKeeper allows for very simple and effective leader election out of the box. This component exploits this election capability in a RoutePolicy to control when and how routes are enabled. This policy would typically be used in fail-over scenarios, to control identical instances of a route across a cluster of Camel based servers. A very common scenario is a simple 'Master-Slave' setup where there are multiple instances of a route distributed across a cluster but only one of them, that of the master, should be running at a time. If the master fails, a new master should be elected from the available slaves and the route in this new master should be started.

The policy uses a common znode path across all instances of the RoutePolicy that will be involved in the election. Each policy writes its id into this node and zookeeper will order the writes in the order it received them. The policy then reads the listing of the node to see what position of its id; this position is used to determine if the route should be started or not. The policy is configured at startup with the number of route instances that should be started across the cluster and if its position in the list is less than this value then its route will be started. For a Master-slave scenario, the route is configured with 1 route instance and only the first entry in the listing will start its route. All policies watch for updates to the listing and if the listing changes they recalculate if their route should be started. The following example uses the node '/someapplication/somepolicy' for the election and is set up to start only the top '1' entries in the node listing i.e. elect a master:

```
ZooKeeperRoutePolicy policy = new ZooKeeperRoutePolicy(
```

```
"zookeeper:localhost:39913/someapp/somepolicy", 1);  
from("direct:policy-controlled").routePolicy(policy)  
  .to("mock:controlled");
```



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 Runtime 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 Talend Runtime 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 and 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.

