JMS

JMS Component

Using ActiveMQ
If you are using Apache ActiveMQ, you should prefer the ActiveMQ component as it has been optimized for ActiveMQ. All of the options and samples on this page are also valid for the ActiveMQ component.
Transacted and caching
See section *Transactions and Cache Levels* below if you are using transactions with JMS as it can impact performance.
Request/Reply over JMS
Make sure to read the section *Request-reply over JMS* further below on this page for important notes about request/reply, as Camel offers a number of options to configure for performance, and clustered environments.

This component allows messages to be sent to (or consumed from) a JMS Queue or Topic. It uses Spring's JMS support for declarative transactions, including 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:

xml

URI Format

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:

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

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

You append query options to the URI using the following format: **`?option=value&option=value&...`**

Notes

Using ActiveMQ
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 - which is a good choice as ActiveMQ rocks , then we recommend that you either:

- Use the ActiveMQ component, which is already optimized to use ActiveMQ efficiently
- Use the **`PoolingConnectionFactory`** in ActiveMQ

Transactions and Cache Levels

transactionCacheLevels
If you are consuming messages and using transactions (**`transacted=true`**) then the default cache level can negatively impact performance. If you are using XA transactions then you cannot cache as it can cause the XA transaction to not work properly.

If you are *not* using XA, then you should consider caching as it speeds up performance, such as setting **`cacheLevelName=CACHE_CONSUMER`**. Through Camel 2.7.x, the default setting for **`cacheLevelName`** is **`CACHE_CONSUMER`**. You will need to explicitly set **`cacheLevelName=CACHE_NONE`**. In Camel 2.8 onward, the default setting for **`cacheLevelName`** is **`CACHE_AUTO`**. This default auto detects the mode and sets the cache level accordingly to:

- **`CACHE_CONSUMER`** when **`transacted=false`**
- **`CACHE_NONE`** when **`transacted=true`**

So you can say the default setting is conservative. Consider using **`cacheLevelName=CACHE_CONSUMER`** if you are using non-XA transactions.

Durable Subscriptions

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 here.

Message Header Mapping

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 and hyphens in the header name as shown below 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:

- Dots are replaced by _**DOT**_ and the replacement is reversed when Camel consume the message
- Hyphen is replaced by _**HYPHEN**_ and the replacement is reversed when Camel consumes the message

Configuration Options

You can configure many different properties on the JMS endpoint which map to properties on the JMSConfiguration POJO.

Mapping to Spring JMS
Many of these properties map to properties on Spring JMS, which Camel uses for sending and receiving messages. Therefore for more information about these properties consult the Spring documentation.

The options are divided into two tables, the first one contains the most common options. The second table contains the less common and more advanced options.

Common Options

confluenceTableSmall

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. From **Camel 2.10.3**: this option can also be used when doing request/reply over JMS. From **Camel 2.16**: there is a new **`replyToConcurrentConsumers`**. See the **`maxMessagesPerTask`** option to control dynamic scaling up/down of threads. When using ActiveMQ beware that the default prefetch policy loads 1000 messages per consumer. See What is the prefetch limit on how to change this.
`disableReplyTo`	`false`	When **`true`**, a producer will behave like a **`InOnly`** exchange with the exception that **`JMSReplyTo`** header is sent out and not be suppressed like in the case of **`InOnly`**. Like **`InOnly`** the producer will not wait for a reply. A consumer with this flag will behave like **`InOnly`**. This feature can be used to bridge **`InOut`** requests to another queue so that a route on the other queue will send it´s response directly back to the original **`JMSReplyTo`**.
`durableSubscriptionName`	`null`	The durable subscriber name for specifying durable topic subscriptions. The **`clientId`** option **must** be configured as well.
`maxConcurrentConsumers`	`1`	Specifies the maximum number of concurrent consumers. From **Camel 2.10.3**: this option can also be used when doing request/reply over JMS. From **Camel 2.16**: there is a new **`replyToMaxConcurrentConsumers`**. See also the **`maxMessagesPerTask`** option to control dynamic scaling up/down of threads. The **`maxMessagesPerTask`** option *must* be set to an integer greater than **`0`** for threads to scale down. Otherwise, the number of threads will remain at **`maxConcurrentConsumers`** until shutdown.
`maxMessagesPerTask`	`-1`	The number of messages a task can receive after which it's terminated. The default, **`-1`**, is unlimited. If you use a range for concurrent consumers e.g., **`concurrentConsumers`** < **`maxConcurrentConsumers`** 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 headers are considered: • **`JMSPriority`** • **`JMSDeliveryMode`** • **`JMSExpiration`**. You can provide some or all of them. If not provided, Camel will fall back to use the values from the endpoint instead. So, when using this option, the headers override the values from the endpoint. The **`explicitQosEnabled`** option, by contrast, will only use options set on the endpoint, and not values from the message header.
`replyTo`	`null`	Provides an explicit **`ReplyTo`** destination, which overrides any incoming value of **`Message.getJMSReplyTo()`**. If you do Request Reply over JMS then **make sure** to read the section *Request-reply over JMS* further below for more details, and the **`replyToType`** option as well.
`replyToConcurrentConsumers`	`1`	**Camel 2.16**: Specifies the default number of concurrent consumers when doing request/reply over JMS.
`replyToMaxConcurrentConsumers`	`1`	**Camel 2.16**: Specifies the maximum number of concurrent consumers when doing request/reply over JMS. See the **`maxMessagesPerTask`** option to control dynamic scaling up/down of threads.
`replyToOnTimeoutMaxConcurrentConsumers`	`1`	**Camel 2.17.2**: Specifies the maximum number of concurrent consumers for continue routing when timeout occurred when using request/reply over JMS.
`replyToOverride`	`null`	**Camel 2.15:** Provides an explicit **`ReplyTo`** destination in the JMS message, which overrides the setting of **`ReplyTo`**. It is useful if you want to forward the message to a remote Queue and receive the reply message from the **`ReplyTo`** destination.
`replyToType`	`null`	**Camel 2.9:** Allows for explicitly specifying which kind of strategy to use for **`replyTo`** queues when doing request/reply over JMS. Possible values are: • **`Temporary`** • **`Shared`** • **`Exclusive`** By default Camel will use **`Temporary`** queues. However if **`replyTo`** has been configured, then **`Shared`** is used by default. This option allows you to use exclusive queues instead of shared queues. For more details see below, and especially the notes about the implications if running in a clustered environment, and the fact that **`Shared`** reply queues has lower performance than its alternatives **`Temporary`** and **`Exclusive`**.
`requestTimeout`	`20000`	**Producer only:** The timeout for waiting for a reply when using the **`InOut`** Exchange Pattern (in milliseconds). From **Camel 2.13/2.12.3**: you can include the header **`CamelJmsRequestTimeout`** to override this endpoint configured timeout value, and thus have per message individual timeout values. See below in section *About time to live* for more details. See also the **`requestTimeoutCheckerInterval`** option.
`selector`	`null`	Sets the JMS Selector, which is an SQL 92 predicate that is used to filter messages within the broker. You may have to encode special characters like '**`=`**' as **`%3D`**. Before **Camel 2.3.0**: this option was not supported in **`CamelConsumerTemplate`**.
`timeToLive`	`null`	When sending messages, specifies the time-to-live of the message (in milliseconds). See below in section *About time to live* for more details.
`transacted`	`false`	Specifies whether to use transacted mode for sending/receiving messages using the **`InOnly`** Exchange Pattern.
`testConnectionOnStartup`	`false`	**Camel 2.1:** Specifies whether to test the connection on startup. This ensures that when Camel starts that all the JMS consumers 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. From **Camel 2.8**: also the JMS producers is tested as well.

Advanced Options

confluenceTableSmall

Option	Default Value	Description
`acceptMessagesWhileStopping`	`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 run-time, while there are still messages enqueued on the queue. If this option is **`false`**, and you stop the JMS route, then messages may be rejected, and the JMS broker would have to attempt re-deliveries, 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 scenario it's recommended this option be set to **`true`**.
`acknowledgementModeName`	`AUTO_ACKNOWLEDGE`	The JMS acknowledgement name, which is one of: • **`SESSION_TRANSACTED`** • **`CLIENT_ACKNOWLEDGE`** • **`AUTO_ACKNOWLEDGE`** • **`DUPS_OK_ACKNOWLEDGE`**.
`acknowledgementMode`	`-1`	The JMS acknowledgement mode defined as an Integer. Allows you to set vendor-specific extensions to the acknowledgment mode. For the regular modes, it is preferable to use the **`acknowledgementModeName`** instead.
`allowNullBody`	`true`	**Camel 2.9.3/2.10.1:** Whether to allow sending messages with no body. If this option is `false` and the message body is null, then an **`JMSException`** is thrown.
`allowReplyManagerQuickStop`	`false`	Whether the **`DefaultMessageListenerContainer`** used in the reply managers for request-reply messaging allow the **`DefaultMessageListenerContainer.runningAllowed`** flag to quick stop in case link **`JmsConfigurationisAcceptMessagesWhileStopping()`** is enabled and CamelContext is currently being stopped. This quick stop ability is enabled by default in the regular JMS consumers but to enable for reply managers you must enable this flag.
`alwaysCopyMessage`	`false`	If **`true`**, Camel will always make a JMS message copy of the message when it is passed to the producer for sending. Copying the message is needed in some situations, such as when a **`replyToDestinationSelectorName`** is set. Camel will set the **`alwaysCopyMessage=true`**, if a **`replyToDestinationSelectorName`** is set.
`asyncConsumer`	`false`	**Camel 2.9:** Whether the **`JmsConsumer`** processes the Exchange asynchronously. If enabled then the **`JmsConsumer`** may pickup the next message from the JMS queue, while the previous message is being processed asynchronously (by the Asynchronous Routing Engine). 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 **`transacted`** has been enabled, then **`asyncConsumer=true`** does not run asynchronously, as transactions must be executed synchronously (Camel 3.0 may support asynchronous transactions).
`asyncStartListener`	`false`	**Camel 2.10:** Whether to startup the **`JmsConsumer`** message listener asynchronously, when starting a route. For example if a **`JmsConsumer`** cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to `true`, you will let routes startup, while the **`JmsConsumer`** connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at `WARN` level, and the consumer will not be able to receive messages. You can then restart the route to retry.
`asyncStopListener`	`false`	**Camel 2.10:** Whether to stop the **`JmsConsumer`** message listener asynchronously, when stopping a route.
`autoStartup`	`true`	Specifies whether the consumer container should auto-startup.
`cacheLevel`		Sets the cache level by ID for the underlying JMS resources. See **`cacheLevelName`** option for more details.

`cacheLevelName`	• `CACHE_AUTO` (Camel >= 2.8.0) • `CACHE_CONSUMER` (Camel <= 2.7.1)	Sets the cache level by name for the underlying JMS resources. Valid values are: • **`CACHE_NONE`** • **`CACHE_CONNECTION`** • **`CACHE_SESSION`** • **`CACHE_CONSUMER`** • **`CACHE_AUTO`** From **Camel 2.8**: the default is **`CACHE_AUTO`**. For **Camel 2.7.1** and older the default is **`CACHE_CONSUMER`**. See the Spring documentation and Transactions Cache Levels for more information.
`connectionFactory`	`null`	The default JMS connection factory to use for the **`listenerConnectionFactory`** and **`templateConnectionFactory`**, if neither is specified.
`consumerType`	`Default`	The Spring JMS listener type to use. A valid value is one of: **`Simple`**, **`Default`** or **`Custom`**. `consumerType` | Spring JMS Listener Type `Default` | **`org.springframework.jms.listener.DefaultMessageListenerContainer`** `Simple` | **`org.springframework.jms.listener.SimpleMessageListenerContainer`** `Custom` | From **Camel 2.10.2**: The **`MessageListenerContainerFactory`** defined by the **`messageListenerContainerFactoryRef`** option which will determine what **`org.springframework.jms.listener.AbstractMessageListenerContainer`** to use. This option was temporarily removed in Camel 2.7 and 2.8 but was re-added in Camel 2.9.
`defaultTaskExecutorType`	`SimpleAsync`	**Camel 2.10.4:** Specifies what default **`TaskExecutor`** type to use in the **`DefaultMessageListenerContainer`**, for both consumer endpoints and the **`ReplyTo`** consumer of producer endpoints. Possible values: **`SimpleAsync`** (uses Spring's SimpleAsyncTaskExecutor) or **`ThreadPool`** (uses Spring's ThreadPoolTaskExecutor with optimal values - cached threadpool-like). If not set, it defaults to the previous behavior, which uses a cached thread pool for consumer endpoints and **`SimpleAsync`** for reply consumers. The use of **`ThreadPool`** 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 **`javax.jms.DeliveryMode`**.
`deliveryPersistent`	`true`	Specifies whether persistent delivery is used by default.
`destination`	`null`	Specifies the JMS Destination object to use on this endpoint.
`destinationName`	`null`	Specifies the JMS destination name to use on this endpoint.
`destinationResolver`	`null`	A pluggable **`org.springframework.jms.support.destination.DestinationResolver`** that allows you to use your own resolver (for example, to lookup the real destination in a JNDI registry).
`disableTimeToLive`	`false`	**Camel 2.8:** Use this option to force disabling time to live. For example when you do request/reply over JMS, then Camel will by default use the **`requestTimeout`** value as time to live on the message being sent. The problem is that the sender and receiver systems have to have their clocks synchronized, so they are in sync. This is not always so easy to archive. So you can use **`disableTimeToLive=true`** to **not** set a time to live value on the sent message. Then the message will not expire on the receiver system. See below in section *About time to live* for more details.
`eagerLoadingOfProperties`	`false`	Enables eager loading of JMS properties as soon as a message is received, which is generally inefficient, because the JMS properties might not be required. But this feature can sometimes catch early any issues with the underlying JMS provider and the use of JMS properties. This feature can also be used for testing purposes, to ensure JMS properties can be understood and handled correctly.

`errorHandler`	`null`	**Camel 2.8.2, 2.9:** Specifies a **`org.springframework.util.ErrorHandler`** to be invoked in case of any uncaught exceptions thrown while processing a **`Message`**. By default these exceptions will be logged at the **`WARN`** level, if no **`errorHandler`** has been configured. From **Camel 2.9.1**: you can configure logging level and whether stack traces should be logged using the below two options. This makes it much easier to configure, than having to code a custom **`errorHandler`**.
`errorHandlerLoggingLevel`	`WARN`	**Camel 2.9.1:** Configures the logging level at which the **`errorHandler`** will log uncaught exceptions.
`errorHandlerLogStackTrace`	`true`	**Camel 2.9.1:** Controls whether a stacktrace should be logged by the default **`errorHandler`**.
`exceptionListener`	`null`	Specifies the JMS Exception Listener that is to be notified of any underlying JMS exceptions.
`explicitQosEnabled`	`false`	Set if the **`deliveryMode`, `priority`** or **`timeToLive`** qualities of service should be used when sending messages. This option is based on Spring's **`JmsTemplate`**. The **`deliveryMode`, `priority`** and **`timeToLive`** options are applied to the current endpoint. This contrasts with the **`preserveMessageQos`** option, which operates at message granularity, reading QoS properties exclusively from the Camel In message headers.
`exposeListenerSession`	`true`	Specifies whether the listener session should be exposed when consuming messages.
`forceSendOriginalMessage`	`false`	**Camel 2.7:** When using **`mapJmsMessage=false`** Camel will create a new JMS message to send to a new JMS destination if you touch the headers (get or set) during the route. Set this option to **`true`** to force Camel to send the original JMS message that was received.
`idleConsumerLimit`	`1`	**Camel 2.8.2**, **2.9**: Specify the limit for the number of consumers that are allowed to be idle at any given time.
`idleTaskExecutionLimit`	`1`	Specifies the limit for idle executions of a receive task, not having received any message within its execution. If this limit is reached, the task will shut down and leave receiving to other executing tasks (in the case of dynamic scheduling; see the **`maxConcurrentConsumers`** setting). There is additional doc available from Spring.
`includeSentJMSMessageID`	`false`	**Camel 2.10.3:** Only applicable when sending to JMS destination using **`InOnly`**, e.g., 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. When set to `true` properties such as **`JMSXAppID`**, and **`JMSXUserID`** etc will be included. **Note:** If you are using a custom **`headerFilterStrategy`** then this option does not apply.
`jmsKeyFormatStrategy`	`default`	Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. **Strategy** / **Description** `default` — Safely marshals dots and hyphens, `'.'` and `'-'`. `passthrough` — Leaves the key as is. Appropriate for use with any JMS broker that tolerates JMS header keys containing illegal characters. **Note**: optionally, a custom implementation can be provided of a **`org.apache.camel.component.jms.JmsKeyFormatStrategy`** and referred to using the # notation.
`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 determines which JMS message type to use for the **`In`** body type. This option will override the default behavior.

jmsOperations	null	Allows you to use your own implementation of the **org.springframework.jms.core.JmsOperations** interface. Camel uses **JmsTemplate** by default. Can be used for testing purpose, but not used much as stated in the spring API docs.
lazyCreateTransactionManager	true	If **true**, Camel will create a **JmsTransactionManager**, if there is no **transactionManager** injected when option **transacted=true**.
listenerConnectionFactory	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 **java.lang.String** etc. See below for more details on how message type mapping works.
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 100% in control how to map to/from a **javax.jms.Message**.
messageIdEnabled	true	When sending, specifies whether message IDs should be added.
messageListenerContainerFactoryRef	null	**Camel 2.10.2:** Registry ID of the **MessageListenerContainerFactory** used to determine what **org.springframework.jms.listener.AbstractMessageListenerContainer** to use to consume messages. Setting this will automatically set **consumerType** to **Custom**.
messageTimestampEnabled	true	Specifies whether time-stamps should be enabled by default on sending messages.
password	null	The password for the connector factory.
priority	4	Values greater than **1** specify the message priority when sending (where **0** is the lowest priority and **9** is the highest). The **explicitQosEnabled** option *must* also be enabled in order for this option to have any effect.
pubSubNoLocal	false	Specifies whether to inhibit the delivery of messages published by its own connection.
receiveTimeout	1000	The timeout for receiving messages (in milliseconds).
recoveryInterval	5000	Specifies the interval between recovery attempts, e.g., when a connection is being refreshed, in milliseconds. The default is **5000** ms.
replyToSameDestinationAllowed	false	**Camel 2.16: Consumer only:** Whether a JMS consumer is allowed to send a reply message to the same destination that the consumer is using to consume from. This prevents an endless loop by consuming and sending back the same message to itself.
replyToCacheLevelName	CACHE_CONSUMER	**Camel 2.9.1:** Sets the cache level by name for the reply consumer when doing request/reply over JMS. This option only applies when using fixed reply queues (not temporary). Camel will by default use: **CACHE_CONSUMER** for exclusive or shared w/ **replyToSelectorName** and **CACHE_SESSION** for shared without **replyToSelectorName**. Some JMS brokers such as IBM WebSphere may require this parameter to be set to **CACHE_NONE** in order to work. **Note:** The value **CACHE_NONE** cannot be used with temporary queues. A higher value, such as **CACHE_CONSUMER** or **CACHE_SESSION**, must be used.
replyToDestinationSelectorName	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).
replyToDeliveryPersistent	true	Specifies whether to use persistent delivery by default for replies.

`requestTimeoutCheckerInterval`	`1000`	**Camel 2.9.2:** 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 option **`requestTimeout`**.
`subscriptionDurable`	`false`	**@deprecated:** Enabled by default, if you specify a **`durableSubscriptionName`** and a **`clientId`**.
`taskExecutor`	`null`	Allows you to specify a custom task executor for consuming messages.
`taskExecutorSpring2`	`null`	**Camel 2.6:** To use when using Spring 2.x with Camel. Allows you to specify a custom task executor for consuming messages.
`templateConnectionFactory`	`null`	The JMS connection factory used for sending messages.
`transactedInOut`	`false`	**@deprecated:** Specifies whether to use transacted mode for sending messages using the **`InOut`** Exchange Pattern. Applies only to producer endpoints. See section Enabling Transacted Consumption for more details.
`transactionManager`	`null`	The Spring transaction manager to use.
`transactionName`	`"JmsConsumer[destinationName]"`	The name of the transaction to use.
`transactionTimeout`	`null`	The timeout value of the transaction (in seconds), if using transacted mode.
`transferException`	`false`	If enabled and you are using Request Reply messaging (**`InOut`**) and an Exchange failed on the consumer side, then the caused **`Exception`** will be send back in response as a **`javax.jms.ObjectMessage`**. If the client is Camel, the returned **`Exception`** is re-thrown. 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 **`Exception`** on the consumer side can be wrapped in an outer exception such as **`org.apache.camel.RuntimeCamelException`** when returned to the producer.
`transferExchange`	`false`	You can transfer the exchange over the wire instead of just the body and headers. The following fields are transferred: In body, Out body, Fault body, In headers, Out headers, Fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at **`WARN`** level. You *must* enable this option on *both* the producer *and* the consumer side, so Camel will know that the payload is an Exchange and not a regular payload.
`transferFault`	`false`	**Camel 2.17:** If enabled and you are using Request Reply messaging (**`InOut`**) and an Exchange failed with a SOAP fault (not exception) on the consumer side, then the fault flag on **`org.apache.camel.Message.isFault()`** will be send back in the response as a JMS header with the key **`JmsConstants.JMS_TRANSFER_FAULT`**. If the client is Camel, the returned fault flag will be set on the **`org.apache.camel.Message.setFault(boolean)`**. You may want to enable this when using Camel components that support faults such as SOAP based such as CXF or spring-ws.
`username`	`null`	The username for the connector factory.
`useMessageIDAsCorrelationID`	`false`	Specifies whether **`JMSMessageID`** should always be used as **`JMSCorrelationID`** for **`InOut`** messages.
`useVersion102`	`false`	**@deprecated (removed from Camel 2.5 onward)** Specifies whether the old JMS API should be used.

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:

confluenceTableSmall

Body Type	JMS Message	Comment
`byte[]`	`javax.jms.BytesMessage`	
`java.io.File`	`javax.jms.BytesMessage`	
`java.io.InputStream`	`javax.jms.BytesMessage`	
`java.io.Reader`	`javax.jms.BytesMessage`	
`java.io.Serializable`	`javax.jms.ObjectMessage`	
`java.nio.ByteBuffer`	`javax.jms.BytesMessage`	
`Map`	`javax.jms.MapMessage`	
`org.w3c.dom.Node`	`javax.jms.TextMessage`	The DOM will be converted to **`String`**.
`String`	`javax.jms.TextMessage`	

When receiving a JMS message, Camel converts the JMS message to the following body type:

confluenceTableSmall

JMS Message	Body Type
`javax.jms.BytesMessage`	`byte[]`
`javax.jms.MapMessage`	`Map<String, Object>`
`javax.jms.ObjectMessage`	`Object`
`javax.jms.TextMessage`	`String`

Disabling Auto-Mapping of JMS Messages

You can use the **`mapJmsMessage`** option to disable the auto-mapping above. If disabled, Camel will not try to map the received JMS message, but instead uses it directly as the payload. This allows you to avoid the overhead of mapping and let Camel just pass through the JMS message. For instance, it even allows you to route **`javax.jms.ObjectMessage`** JMS messages with classes you do **not** have on the classpath.

Using a custom MessageConverter

You can use the **`messageConverter`** option to do the mapping yourself in a Spring **`org.springframework.jms.support.converter.MessageConverter`** class.

For example, in the route below we use a custom message converter when sending a message to the JMS order queue:

java

You can also use a custom message converter when consuming from a JMS destination.

Controlling the Mapping Strategy Selected

You can use the **`jmsMessageType`** option on the endpoint URL to force a specific message type for all messages. In the route below, we poll files from a folder and send them as **`javax.jms.TextMessage`** as we have forced the JMS producer endpoint to use text messages:

java

You can also specify the message type to use for each message by setting the header with the key **`CamelJmsMessageType`**. For example:

java

The possible values are defined in the `enum` class **`org.apache.camel.jms.JmsMessageType`**.

Message Format When Sending

An exchange sent via JMS must conform to the JMS Message spec. Camel therefore applies various translation and validation rules to both key names and key values of **`exchange.in.headers`**.

The following rules are applied to the *key names* of **`exchange.in.headers`**:

- Keys starting with **`JMS`** or **`JMSX`** are reserved.
- Key names must be literals or valid Java identifiers.
- Dot and hyphen characters are replaced (and the reverse when consuming) as follows:
    - The character **`'.'`** is replaced with the sequence **`_DOT_`**. The reverse replacement is applied when Camel consumes a message.
    - The character **`'-'`** is replaced with the sequence **`_HYPHEN_`**. The reverse replacement is applied when Camel consumes a message.

- The option **`jmsKeyFormatStrategy`** can be used to specify a custom key formatting strategy.

The following rules are applied to the *key values* of **`exchange.in.headers`**:

- Values must be either a primitive type or of its corresponding Java object type, e.g., **`Integer`**, **`Long`** or **`Character`**.
- The types **`String`**, **`CharSequence`**, **`Date`**, **`BigDecimal`** and **`BigInteger`** are all converted to their string representation.
- All other types will result in the key value being discarded.

If a header value is discarded Camel will log the incident using logging category **`org.apache.camel.component.jms.JmsBinding`** at the **`DEBUG`** logging level. For example:

text

Message Format When Receiving

Camel adds the following properties to the **`Exchange`** when it receives a message:

confluenceTableSmall

Property	Type	Description
`org.apache.camel.jms.replyDestination`	`javax.jms.Destination`	The reply destination.

Camel adds the following JMS properties to the In message headers when it receives a JMS message:

confluenceTableSmall

Header	Type	Description
`JMSCorrelationID`	`String`	The JMS correlation ID.
`JMSDeliveryMode`	`int`	The JMS delivery mode.
`JMSDestination`	`javax.jms.Destination`	The JMS destination.
`JMSExpiration`	`long`	The JMS expiration.
`JMSMessageID`	`String`	The JMS unique message ID.
`JMSPriority`	`int`	The JMS priority (with 0 as the lowest priority and 9 as the highest).
`JMSRedelivered`	`boolean`	Is the JMS message redelivered.
`JMSReplyTo`	`javax.jms.Destination`	The JMS reply-to destination.
`JMSTimestamp`	`long`	The JMS timestamp.
`JMSType`	`String`	The JMS type.
`JMSXGroupID`	`String`	The JMS group ID.

As all the above information is standard JMS you can check the JMS documentation for further details.

Using Camel to Send and Receive Messages Using `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 (MEP)
- 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`** or **`explicitQosEnabled`**.

All this can be a tad complex to understand and configure to support your use case.

JmsProducer

The **`JmsProducer`** behaves as follows, depending on configuration:

confluenceTableSmall

Exchange Pattern	Other options	Description
`InOut`	-	Camel will expect a reply, set a temporary **`JMSReplyTo`**, and after sending the message, it will start to listen for the reply message on the temporary queue.

`InOut`	`JMSReplyTo` is set	Camel will expect a reply and, after sending the message, it will start to listen for the reply message on the specified **`JMSReplyTo`** queue.
`InOnly`	-	Camel will send the message and **not** expect a reply.
`InOnly`	`JMSReplyTo` is set	By default, Camel discards the **`JMSReplyTo`** destination and clears the **`JMSReplyTo`** header before sending the message. Camel then sends the message and does *not* expect a reply. Camel logs this in the log at **`WARN`** level (changed to **`DEBUG`** level from **Camel 2.6** on). You can use **`preserveMessageQuo=true`** to instruct Camel to keep the **`JMSReplyTo`**. In all situations the **`JmsProducer`** does **not** expect any reply and thus continue after sending the message.

JmsConsumer

The `JmsConsumer` behaves as follows, depending on configuration:

confluenceTableSmall

Exchange Pattern	Other options	Description
`InOut`	-	Camel will send the reply back to the **`JMSReplyTo`** queue.
`InOnly`	-	Camel will not send a reply back, as the pattern is **`InOnly`**.
-	`disableReplyTo=true`	This option suppresses replies.

So 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:

java

Computing the Destination at Runtime

If you need to send messages to a lot of different JMS destinations, it makes sense to reuse a JMS endpoint and specify the real destination in a message header. This allows Camel to reuse the same endpoint, but send to different destinations. This greatly reduces the number of endpoints created and economizes on memory and thread resources.

You can specify the destination in the following headers:

confluenceTableSmall

Header	Type	Description
`CamelJmsDestination`	`javax.jms.Destination`	A destination object.
`CamelJmsDestinationName`	`String`	The destination name.

For example, the following route shows how you can compute a destination at run time and use it to override the destination appearing in the JMS URL:

java

The queue name, `dummy`, is just a placeholder. It must be provided as part of the JMS endpoint URL, but it will be ignored in this example.

In the **`computeDestination`** bean, specify the real destination by setting the **`CamelJmsDestinationName`** header as follows:

java

Then Camel will read this header and use it as the destination instead of the one configured on the endpoint. So, in this example Camel sends the message to **`activemq:queue:order:2`**, assuming the **`id`** value was **`2`**.

If both the `CamelJmsDestination` and the **`CamelJmsDestinationName`** headers are set **`CamelJmsDestination`** will take priority. Note that the JMS producer removes both **`CamelJmsDestination`** and **`CamelJmsDestinationName`** headers from the exchange and does not propagate them to the created JMS message. This prevents accidental routing loops in scenarios where a message is forwarded to another JMS endpoint.

Configuring Different JMS Providers

A JMS provider can be configured in Spring XML as follows:

An unlimited number of JMS component instance can be created provided each has a *unique value for its* **`id`** *attribute. The preceding example configures an activemq component. You could do the same to configure MQSeries, TibCo, BEA, Sonic etc.*

Once named a JMS component can be referenced from an endpoint's URI. For example, given the component name `activemq` a URI can reference the component using the format **`activemq:[queue:|topic:]destinationName`**. The same approach applies to all JMS providers. This is achieved by the **`SpringCamelContext`** lazily fetching components from the spring context for the scheme name referenced in the Endpoint URIs then having the Component resolve the endpoint URI itself.

Using JNDI to Find the Connection Factory

If you are using a J2EE container, you might need to look up JNDI to find the JMS **`connectionFactory`** rather than use the usual **`<bean>`** mechanism in Spring. You can do this using Spring's factory bean or the new Spring XML namespace. For example:

xml

See The jee schema in the Spring reference documentation for more details about JNDI lookup.

Concurrent Consuming

A common requirement with JMS is to consume messages concurrently in multiple threads in order to make an application more responsive. You can set the **`concurrentConsumers`** option to specify the number of threads servicing the JMS endpoint, as follows:

java

You can configure this option in one of the following ways:

- On the **`JmsComponent`**
- On the endpoint URI
- By invoking **`setConcurrentConsumers()`** directly on the **`JmsEndpoint`**.

Concurrent Consuming with `asyncConsumer`

Notice that each concurrent consumer will only pickup the next available message from the JMS broker, when the current message has been fully processed. You can set the option **`asyncConsumer=true`** to let the consumer pickup the next message from the JMS queue, while the previous message is being processed asynchronously (by the Asynchronous Routing Engine). See more details in the table on top of the page about the **`asyncConsumer`** option.

java

Request-Reply over JMS

Camel supports Request Reply over JMS. In essence the MEP of the Exchange should be **`InOut`** when you send a message to a JMS queue.

Camel offers a number of options to configure request/reply over JMS that influence performance and clustered environments. The table below summaries the options.

confluenceTableSmall

Option	Performance	Cluster	Description
`Temporary`	Fast	Yes	A temporary queue is used as reply queue, and automatic created by Camel. To use this do **not** specify a **`replyTo`** queue name. And you can optionally configure **`replyToType=Temporary`** to make it stand out that temporary queues are in use.
`Shared`	Slow	Yes	A shared persistent queue is used as reply queue. The queue must be created beforehand, although some brokers can create them on the fly such as Apache ActiveMQ. To use this you must specify the **`replyTo`** queue name. And you can optionally configure **`replyToType=Shared`** to make it stand out that shared queues are in use. A shared queue can be used in a clustered environment with multiple nodes running this Camel application at the same time. All using the same shared reply queue. This is possible because JMS Message selectors are used to correlate expected reply messages; this impacts performance though. JMS Message selectors is slower, and therefore not as fast as **`Temporary`** or **`Exclusive`** queues. See further below how to tweak this for better performance.
`Exclusive`	Fast	No (*Yes)	An exclusive persistent queue is used as reply queue. The queue must be created beforehand, although some brokers can create them on the fly such as Apache ActiveMQ. To use this you must specify the replyTo queue name. And you **must** configure **`replyToType=Exclusive`** to instruct Camel to use exclusive queues, as `Shared` is used by default, if a **`replyTo`** queue name was configured. When using exclusive reply queues, then JMS Message selectors are **not** in use, and therefore other applications must not use this queue as well. An exclusive queue **cannot** be used in a clustered environment with multiple nodes running this Camel application at the same time; as we do not have control if the reply queue comes back to the same node that sent the request message; that is why shared queues use JMS Message selectors to make sure of this. **Though** if you configure each Exclusive reply queue with an unique name per node, then you can run this in a clustered environment. As then the reply message will be sent back to that queue for the given node, that awaits the reply message.
`concurrentConsumers`	Fast	Yes	**Camel 2.10.3:** Allows to process reply messages concurrently using concurrent message listeners in use. You can specify a range using the **`concurrentConsumers`** and **`maxConcurrentConsumers`** options. **Note:** That using `Shared` reply queues may not work as well with concurrent listeners, so use this option with care.
`maxConcurrentConsumers`	Fast	Yes	**Camel 2.10.3:** Allows to process reply messages concurrently using concurrent message listeners in use. You can specify a range using the **`concurrentConsumers`** and **`maxConcurrentConsumers`** options. **Note:** That using `Shared` reply queues may not work as well with concurrent listeners, so use this option with care.

The **`JmsProducer`** detects the **`InOut`** and provides a **`JMSReplyTo`** header with the reply destination to be used. By default Camel uses a temporary queue, but you can use the **`replyTo`** option on the endpoint to specify a fixed reply queue (see more below about fixed reply queue).

Camel will automatic setup a consumer which listen on the reply queue, so you should **not** do anything. This consumer is a Spring **`DefaultMessageListenerContainer`** which listen for replies. However it's fixed to **`1`** concurrent consumer. That means replies will be processed in sequence as there are only 1 thread to process the replies. If you want to process replies faster, then we need to use concurrency. But **not** using the **`concurrentConsumer`** option. We should use the `threads` from the Camel DSL instead, as shown in the route below:

Instead of using threads, then use **`concurrentConsumers`** option if using Camel 2.10.3 or greater. See below for details.
java

In this route we instruct Camel to route replies asynchronously using a thread pool with 5 threads.

From **Camel 2.10.3**: you can now configure the listener to use concurrent threads using the **`concurrentConsumers`** and **`maxConcurrentConsumers`** options. This allows you to easier configure this in Camel as shown below:

java

Request-Reply over JMS Using a Shared Fixed Reply Queue

If you use a fixed reply queue when doing Request Reply over JMS as shown in the example below, then pay attention.

In this example the fixed reply queue named "bar" is used. By default Camel assumes the queue is shared when using fixed reply queues, and therefore it uses a **`JMSSelector`** to only pickup the expected reply messages (eg based on the **`JMSCorrelationID`**). See next section for exclusive fixed reply queues. That means its not as fast as temporary queues. You can speedup how often Camel will pull for reply messages using the **`receiveTimeout`** option. By default its **`1000ms`**. So to make it faster you can set it to **`250ms`** to pull **`4`** times per second as shown:

java

Notice this will cause the Camel to send pull requests to the message broker more frequent, and thus require more network traffic. It's generally recommended that temporary queues be used where possible.

Request-Reply over JMS Using an Exclusive Fixed Reply Queue

Available as of Camel 2.9

In the previous example, Camel would anticipate the fixed reply queue named `bar` was shared, and thus it uses a **`JMSSelector`** to only consume reply messages which it expects. However there is a drawback doing this as JMS selectos is slower. Also the consumer on the reply queue is slower to update with new JMS selector ids. In fact it only updates when the **`receiveTimeout`** option times out, which by default is 1 second. So in theory the reply messages could take up till about 1 sec to be detected. On the other hand if the fixed reply queue is exclusive to the Camel reply consumer, then we can avoid using the JMS selectors, and thus be more performant. In fact as fast as using temporary queues. So in **Camel 2.9** onward we introduced the **`ReplyToType`** option which you can configure to **`Exclusive`** to tell Camel that the reply queue is exclusive as shown in the example below:

java

Mind that the queue must be exclusive to each and every endpoint. So if you have two routes, then they each need an unique reply queue as shown in the next example:

java

The same applies if you run in a clustered environment. Then each node in the cluster must use an unique reply queue name. As otherwise each node in the cluster may pickup messages which was intended as a reply on another node. For clustered environments its recommended to use shared reply queues instead.

Synchronizing Clocks Between Senders and Receivers

When doing messaging between systems, its desirable that the systems have synchronized clocks. For example when sending a JMS message, then you can set a time to live value on the message. Then the receiver can inspect this value, and determine if the message is already expired, and thus drop the message instead of consume and process it. However this requires that both sender and receiver have synchronized clocks. If you are using ActiveMQ then you can use the timestamp plugin to synchronize clocks.

About Time To Live

Read first above about synchronized clocks.

When you do request/reply, **`InOut`**, over JMS Camel uses a timeout on the sender side, which is default 20 seconds, taken from the **`requestTimeout`** option. You can control this by setting a higher/lower value. However, the time to live value is still set on the JMS message being sent. This therefore requires that system clocks be synchronized between the systems. If they are not, then you may want to disable the time to live value being set. This is now possible using the **`disableTimeToLive`** option from **Camel 2.8** onward. So if you set this option to **`disableTimeToLive=true`**, then Camel does **not** set any time to live value when sending JMS messages. **But** the request timeout is still active. So for example if you do request/reply over JMS and have disabled time to live, then Camel will still use a timeout by 20 seconds (the **`requestTimeout`** option). That option can of course also be configured. So the two options **`requestTimeout`** and **`disableTimeToLive`** gives you fine grained control when doing request/reply.

From **Camel 2.13/2.12.3**: you can provide a header in the message to override and use as the request timeout value instead of the endpoint configured value. For example:

java

In the route above we have a endpoint configured **`requestTimeout`** of 30 seconds. So Camel will wait up till 30 seconds for that reply message to come back on the bar queue. If no reply message is received then a **`org.apache.camel.ExchangeTimedOutException`** is set on the Exchange and Camel continues routing the message, which would then fail due the exception, and Camel's error handler reacts.

If you want to use a per message timeout value, you can set the header with key **`org.apache.camel.component.jms.JmsConstants#JMS_REQUEST_TIMEOUT`** which has constant value **`CamelJmsRequestTimeout`** with a timeout value as long type.

For example we can use a bean to compute the timeout value per individual message, such as calling the `"whatIsTheTimeout"` method on the service bean as shown below:

java

When you do fire and forget (**`InOut`**) over JMS Camel will, by default, **not** set a time to live value on the message. The value can be configured using the **`timeToLive`** option. For example to indicate a 5 sec., you set **`timeToLive=5000`**. The option **`disableTimeToLive`** can be used to force disabling the time to live, also for **`InOnly`** messaging. The **`requestTimeout`** option is not being used for **`InOnly`** messaging.

Enabling Transacted Consumption

transactedConsumption

A common requirement is to consume from a queue in a transaction and then process the message using the Camel route. To do this, just ensure that you set the following properties on the component/endpoint:

- `transacted = true`
- `transactionManager` = <*SomeTransactionManager*> (typically the JmsTransactionManager)

See the Transactional Client EIP pattern for further details.

Transactions and [Request Reply] over JMS
When using Request Reply over JMS you cannot use a single transaction; JMS will not send any messages until a commit is performed, so the server side won't receive anything at all until the transaction commits. Therefore to use Request Reply you must commit a transaction after sending the request and then use a separate transaction for receiving the response.

To address this issue the JMS component uses different properties to specify transaction use for oneway messaging and request reply messaging:

- The **`transacted`** property applies **only** to the **`InOnly`** message Exchange Pattern (MEP).
- The **`transactedInOut`** property applies to the **`InOut`** (Request Reply) message Exchange Pattern (MEP). If you want to use transactions with the **`InOut`** (Request Reply) message Exchange Pattern (MEP), you *must* set **`transactedInOut=true`**.

Available as of Camel 2.10

You can leverage the DMLC transacted session API using the following properties on component/endpoint:

- `transacted = true`
- `lazyCreateTransactionManager = false`

The benefit of doing so is that the `cacheLevel` setting will be honored when using local transactions without a configured TransactionManager. When a TransactionManager is configured, no caching happens at DMLC level and its necessary to rely on a pooled connection factory. For more details about this kind of setup see here and here.

Using `JMSReplyTo` For Late Replies

When using Camel as a JMS listener, it sets an Exchange property with the value of the ReplyTo **`javax.jms.Destination`** object, having the key **`ReplyTo`**. You can obtain this **`Destination`** as follows:

java

And then later use it to send a reply using regular JMS or Camel.

java

A different solution to sending a reply is to provide the **`replyDestination`** object in the same Exchange property when sending. Camel will then pick up this property and use it for the real destination. The endpoint URI must include a dummy destination, however.

Example:

java

Using a Request Timeout

In the sample below we send a Request Reply style message Exchange (we use the **`requestBody`** method = **`InOut`**) to the slow queue for further processing in Camel and we wait for a return reply:

Examples

JMS is used in many examples for other components as well. But we provide a few samples below to get started.

Receiving from JMS

In the following sample we configure a route that receives JMS messages and routes the message to a POJO:

java

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:

java

Sending to 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`**:

java

Using Annotations

Camel also has annotations so you can use POJO Consuming and POJO Producing.

Spring DSL Example

The preceding examples use the Java DSL. Camel also supports Spring XML DSL. Here is the big spender sample using Spring DSL:

xml

Other Examples

JMS appears in many of the examples for other components and EIP patterns, as well in this Camel documentation. So feel free to browse the documentation. If you have time, check out the this tutorial that uses JMS but focuses on how well Spring Remoting and Camel works together Tutorial-JmsRemoting.

Using JMS as a Dead Letter Queue Storing Exchange

Normally, when using JMS as the transport, it only transfers the body and headers as the payload. If you want to use JMS with a Dead Letter Channel, using a JMS queue as the Dead Letter Queue, then normally the caused Exception is not stored in the JMS message. You can, however, use the **`transferExchange`** option on the JMS dead letter queue to instruct Camel to store the entire Exchange in the queue as a **`javax.jms.ObjectMessage`** that holds a **`org.apache.camel.impl.DefaultExchangeHolder`**. This allows you to consume from the Dead Letter Queue and retrieve the caused exception from the Exchange property with the key **`Exchange.EXCEPTION_CAUGHT`**.

Example:

java

Then you can consume from the JMS queue and analyze the problem:

java

Using JMS as a Dead Letter Channel for Storing Error Only

You can use JMS to store the cause error message or to store a custom body, which you can initialize yourself. The following example uses the Message Translator EIP to do a transformation on the failed exchange before it is moved to the JMS dead letter queue:

java

Here we only store the original cause error message in the transform. You can, however, use any Expression to send whatever you like. For example, you can invoke a method on a Bean or use a custom processor.

Sending an `InOnly` Message and Keeping the `JMSReplyTo` Header

When sending to a JMS destination using **camel-jms** the producer will use the MEP to detect if it's **`InOnly`** or **`InOut`** messaging. However, there can be times where you want to send an **`InOnly`** message but keeping the **`JMSReplyTo`** header. To do so you have to instruct Camel to keep it, otherwise the **`JMSReplyTo`** header will be dropped.

For example to send an **`InOnly`** message to the **`foo`** queue, but with a **`JMSReplyTo`** with bar queue you can do as follows:

java

Note: we use **`preserveMessageQos=true`** to instruct Camel to keep the **`JMSReplyTo`** header.

Setting JMS Provider Options on the Destination

Some JMS providers, like IBM's WebSphere MQ need options to be set on the JMS destination. For example, you may need to specify the **`targetClient`** option. Since targetClient is a WebSphere MQ option and not a Camel URI option, you need to set that on the JMS destination name like so:

java

Some versions of Websphere MQ do not accept this option on the destination name. The following exception is raised when this happens:

```
com.ibm.msg.client.jms.DetailedJMSException: JMSCC0005: The specified value 'MY_QUEUE?
targetClient=1' is not allowed for 'XMSC_DESTINATION_NAME'
```

A workaround is to use a custom **`DestinationResolver`**:

java

Endpoint See Also

- Transactional Client
- Bean Integration
- Tutorial-JmsRemoting
- JMSTemplate gotchas