Async

Async

Available as of Camel 2.0

The asynchronous API in Camel have been rewritten for Camel 2.0, and the information on this page applies for Camel 2.0 and later.

The Async API in Camel is primarily divided in two areas:

1. Initiating an Async messaging from the client
2. Turning a route into Async using the **threads** DSL

Before we look at these two areas we start with a bit of background information and looks at the concept from at a higher level using diagrams. Then we check out the first area how a client can initiate an Async message exchange and we also throw in the synchronous message exchange in the mix as well so we can compare and distill the difference. And finally we turn our attention towards the last area the new **threads** DSL and what it can be used for.

Background

The new Async API in Camel 2.0 leverages in much greater detail the Java Concurrency API and its support for executing tasks asynchronous. Therefore the Camel Async API should be familiar for users with knowledge of the Java Concurrency API.

A few concepts to master

When doing messaging there are a few aspects to keep in mind. First of all a caller can initiate a message exchange as either:

- Request only
- Request Reply

Request only is when the caller sends a message but do **not** expect any reply. This is also known as fire and forget or event message. The Request Reply is when the caller sends a message and then **waits for a reply**. This is like the HTTP protocol that we use every day when we surf the web. We send a request to fetch a web page and wait until the reply message comes with the web content.

In Camel a message is labeled with a Message Exchange Pattern that labels whether its a request only or request reply message. Camel uses the JBI term for this an uses **`InOnly`** for the request only, and **`InOut`** for the request reply.

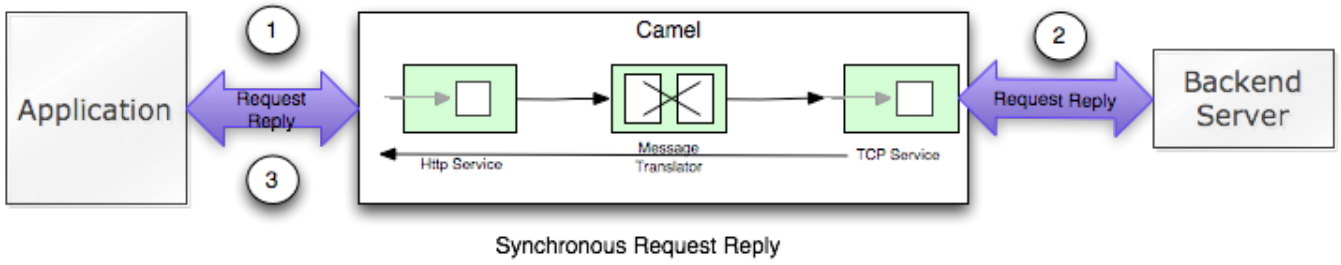
For all message exchange they can be executed either:

- synchronous
- asynchronous

Synchronous Request Reply

A synchronous exchange is defined as the caller sends a message and waits until its complete before continuing.

This is illustrated in the diagram below:



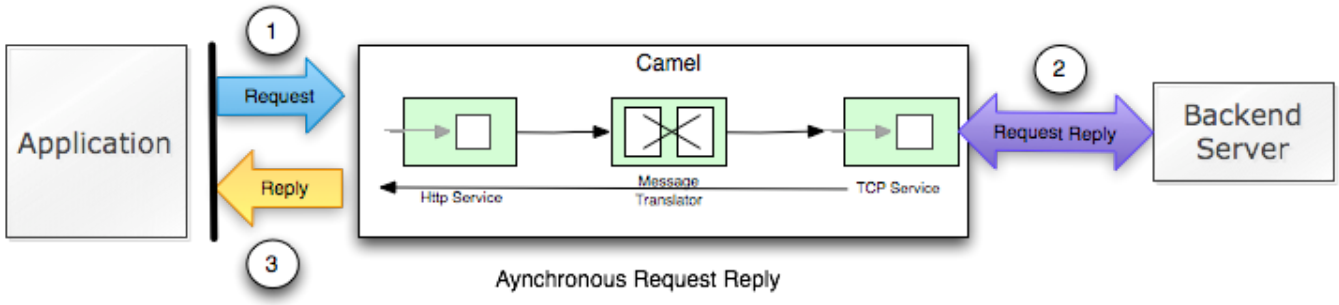
Synchronous Request Reply

1. The client sends a sync Request Reply message over HTTP to Camel. The client application will wait for the response that Camel routes and processes.
2. The message invokes an external TCP service using synchronous Request Reply. The client application still waits for the response.
3. The response is send back to the client.

Asynchronous Request Reply

On the other hand the asynchronous version is where the caller sends a message to an Endpoint and then returns immediately back to the caller. The message however is processed in another thread, the asynchronous thread. Then the caller can continue doing other work and at the same time the asynchronous thread is processing the message.

This is illustrated in the diagram below:



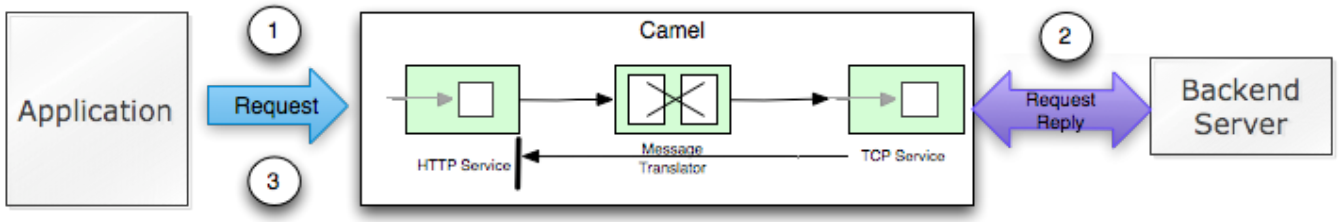
Aynchronous Request Reply

1. The client sends an AsyncRequest Reply message over HTTP to Camel. The control is immediately returned to the client application, that can continue and do other work while Camel routes the message.
2. Camel invokes an external TCP service using synchronous Request Reply. The client application can do other work simultaneously.
3. The client wants to get the reply so it uses the Future handle it got as *response* from step 1. With this handle it retrieves the reply, wait if necessary if the reply is not ready.

Synchronous Request Only

You can also do synchronous Request only with Camel. The client sends a message to Camel in which a reply is not expected. However the client still waits until the message is processed completely.

This is illustrated in the diagram below:



Synchronous Request Only

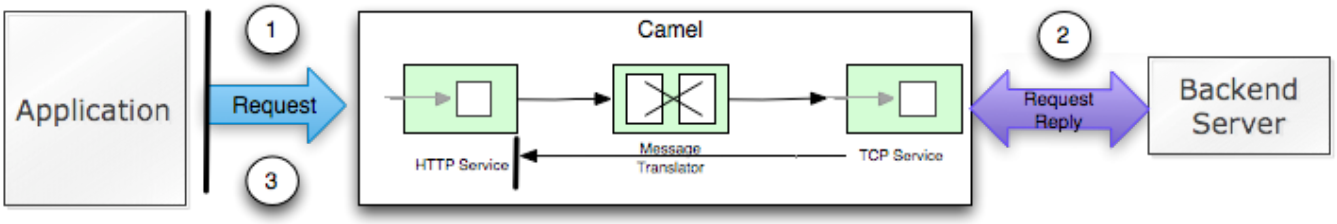
1. The client sends a Request only and we can still use HTTP despite http being Request Reply by nature.
2. Camel invokes an external TCP service using synchronous Request Reply. The client application is still waiting.
3. The message is processed completely and the control is returned to the client.

So why do you want to use synchronous Request Only? Well if you want to know whether the message was processed successfully or not before continuing. With synchronous it allows you to wait while the message is being processed. In case the processing was successful the control is returned to the client with no notion of error. In case of failure the client can detect this as an exception is thrown. (and **`exchange.isFailed()`** returns **`true`).**

Asynchronous Request Only

As opposed to the synchronous Request Only the Async counter part will **not** wait for the processing of the message to complete. In this case the client can immediately continue doing other work while the message is being routed and processed in Camel.

This is illustrated in the diagram below:



Asynchronous Request Only

1. The client sends a Request only and we can still use HTTP despite HTTP being Request Reply by nature. The control is immediately returned to the client application, that can continue and do other work while Camel routes the message.
2. Camel invokes an external TCP service using synchronous Request Reply. The client application can do other work simultaneously.
3. The message completes but no result is returned to the client.

Note: As Camel always returns a **`Future`** handle for Async messaging to the client. The client can use this handler to get hold of the status of the processing whether the task is complete or an Exception occurred during processing. Note that the client is not required to do so, its perfect valid to just ignore the Future handle.
Knowing if an Asynchronous Request Only failed
In case you want to know whether the Async Request Only failed, then you can use the **`Future`** handle and invoke **`get()`** and if it throws a **`ExecutionException`** then the processing failed. The caused exception is wrapped. You can invoke **`isDone()`** first to test whether the task is done or still in progress. Otherwise invoking **`get()`** will wait until the task is done.

With these diagrams in mind lets turn out attention to the Async API and how to use it with Camel.

1) The Async Client API

Camel provides the Async Client API in the ProducerTemplate where we have added about ten new methods to Camel 2.0.

We have listed the most important in the table below:

Method	Returns	Description
`setExecutorService`	`void`	Is used to set the Java **`ExecutorService`**. Camel will by default provide a **`ScheduledExecutorService`** with 5 thread in the pool.
`asyncSend`	`Future<Exchange>`	Is used to send an async exchange to a Camel Endpoint. Camel will immediately return control to the caller thread after the task has been submitted to the executor service. This allows you to do other work while Camel processes the exchange in the other async thread.
`asyncSendBody`	`Future<Object>`	As above but for sending body only. This is a request only messaging style so no reply is expected. Uses the **`InOnly`** exchange pattern.
`asyncRequestBody`	`Future<Object>`	As above but for sending body only. This is a Request Reply messaging style so a reply is expected. Uses the **`InOut`** exchange pattern.
`extractFutureBody`	`T`	Is used to get the result from the asynchronous thread using the Java Concurrency Future handle.

The **`asyncSend`** and **`asyncRequest`** methods return a Future handle. This handle is what the caller must use later to retrieve the asynchronous response. You can do this by using the **`extractFutureBody`** method, or just use plain Java but invoke **`get()`** on the **`Future`** handle.

The Async Client API with callbacks

In addition to the Client API from above Camel provides a variation that uses callbacks when the message Exchange is done.

Method	Returns	Description
`asyncCallback`	`Future<Exchange>`	In addition a callback is passed in as a parameter using the **`org.apache.camel.spi.Synchronization`** Callback. The callback is invoked when the message exchange is done.
`asyncCallbackSendBody`	`Future<Object>`	As above but for sending body only. This is a request only messaging style so no reply is expected. Uses the **`InOnly`** exchange pattern.

`asyncCallbackRequestBody`	`Future<Object>`	As above but for sending body only. This is a Request Reply messaging style so a reply is expected. Uses the **`InOut`** exchange pattern.

These methods also returns the **`Future`** handle in case you need them. The difference is that they invokes the callback as well when the Exchange is done being routed.

The Future API

The **`java.util.concurrent.Future`** API have among others the following methods:

Method	Returns	Description
`isDone`	`boolean`	Returns a boolean whether the task is done or not. Will even return **`true`** if the tasks failed due to an exception thrown.
`get()`	`Object`	Gets the response of the task. In case of an exception was thrown the **`java.util.concurrent.ExecutionException`** is thrown with the caused exception.

Example: Asynchronous Request Reply

Suppose we want to call a HTTP service but it is usually slow and thus we do not want to block and wait for the response, as we can do other important computation. So we can initiate an Async exchange to the HTTP endpoint and then do other stuff while the slow HTTP service is processing our request. And then a bit later we can use the **`Future`** handle to get the response from the HTTP service. Yeah nice so lets do it:

First we define some routes in Camel. One for the HTTP service where we simulate a slow server as it takes at least 1 second to reply. And then other route that we want to invoke while the HTTP service is on route. This allows you to be able to process the two routes simultaneously:{snippet:id=e1|lang=java|url=camel/trunk/tests/camel-itest/src/test/java/org/apache/camel/itest/async/HttpAsyncTest.java}And then we have the client API where we call the two routes and we can get the responses from both of them. As the code is based on unit test there is a bit of mock in there as well:{snippet:id=e2|lang=java|url=camel/trunk/tests/camel-itest/src/test/java/org/apache/camel/itest/async/HttpAsyncTest.java}All together it should give you the basic idea how to use this Async API and what it can do.

Example: Synchronous Request Reply

This example is just to a pure synchronous version of the example from above that was Async based.

The route is the same, so its just how the client initiate and send the messages that differs:{snippet:id=e2|lang=java|url=camel/trunk/tests/camel-itest/src/test/java/org/apache/camel/itest/async/HttpSyncTest.java}

Using the Async API with callbacks

Suppose we want to call a HTTP service but it is usually slow and thus we do not want to block and wait for the response, but instead let a callback gather the response. This allows us to send multiple requests without waiting for the replies before we can send the next request.

First we define a route in Camel for the HTTP service where we simulate a slow server as it takes at least 1 second to reply.{snippet:id=e1|lang=java|url=camel/trunk/tests/camel-itest/src/test/java/org/apache/camel/itest/async/HttpAsyncCallbackTest.java}Then we define our callback where we gather the responses. As this is based on an unit test it just gathers the responses in a list. This is a shared callback we use for every request we send in, but you can use your own individual or use an anonymous callback. The callback supports different methods, but we use **`onDone`** that is invoked regardless if the Exchange was processed successfully or failed. The **`org.apache.camel.spi.Synchronization`** API provides fine grained methods for **`onCompletion`** and **`onFailure`** for the two situations.{snippet:id=e2|lang=java|url=camel/trunk/tests/camel-itest/src/test/java/org/apache/camel/itest/async/HttpAsyncCallbackTest.java}And then we have the client API where we call the HTTP service using **`asyncCallback`** 3 times with different input. As the invocation is Async the client will send 3 requests right after each other, so we have 3 concurrent exchanges in progress. The response is gathered by our callback so we do not have to care how to get the response.{snippet:id=e3|lang=java|url=camel/trunk/tests/camel-itest/src/test/java/org/apache/camel/itest/async/HttpAsyncCallbackTest.java}

Using the Async API with the Camel classic API

When using the Camel API to create a producer and send an Exchange we do it like this:
javaEndpoint endpoint = context.getEndpoint("http://slowserver.org/myservice"); Exchange exchange = endpoint.createExchange(); exchange.getIn().setBody("Order ABC"); // create a regular producer Producer producer = endpoint.createProducer(); // send the exchange and wait for the reply as this is synchronous producer.process(exchange);

But to do the same with Async we need a little help from a helper class, so the code is:
javaEndpoint endpoint = context.getEndpoint("http://slowserver.org/myservice"); Exchange exchange = endpoint.createExchange(); exchange.getIn().setBody("Order ABC"); // create a regular producer Producer producer = endpoint.createProducer(); // normally you will use a shared exectutor service with pools ExecutorService executor = Executors.newSingleThreadExecutor(); // send it async with the help of this

helper Future<Exchange> future = AsyncProcessorHelper.asyncProcess(executor, producer, exchange); // here we got the future handle and we can do other stuff while the exchange is being routed in the other asynchronous thread ... // and to get the response we use regular Java Concurrency API Exchange response = future.get();

2) Using the Threads DSL

In Camel 2.0 the `threads` DSL replaces the old `thread` DSL.

Camel 2.0 to 2.3 behavior

The `threads` DSL leverages the JDK concurrency framework for multi threading. It can be used to turn a synchronous route into Async. What happens is that from the point forwards from **`threads`** the messages is routed asynchronous in a new thread. The caller will either wait for a reply if a reply is expected, such as when we use Request Reply messaging. Or the caller will complete as well if no reply was expected such as Request Only messaging.

From Camel 2.4 on behavior

The `threads` DSL leverages the JDK concurrency framework for multi threading. It can be used to turn a synchronous route into Async. What happens is that from the point forwards from **`threads`** the messages is routed asynchronous in a new thread. Camel leverages the asynchronous routing engine, which was re-introduced in Camel 2.4, to continue routing the Exchange asynchronously.

The `threads` DSL supports the following options:

Option	Default	Description
`poolSize`	`10`	A number to indicate the core pool size of the underlying Java **`ExecutorService`** that is actually doing all the heavy lifting of handling Async tasks and correlate replies etc.
`maxPoolSize`		A number to indicate the maximum pool size of the of the underlying Java **`ExecutorService`**
`keepAliveTime`		A number to indicate how long to keep inactive threads alive
`timeUnit`		Time unit for the **`keepAliveTime`** option
`maxQueueSize`		A number to indicate the maximum number of tasks to keep in the worker queue for the underlying Java **`ExecutorService`**
`threadName`		To use a custom thread name pattern. See Threading Model for more details.
`rejectedPolicy`		How to handle rejected tasks. The following options are supported: • **`Abort`** • **`CallerRuns`** • **`Discard`** • **`DiscardOldest`** See below for more details.
`callerRunsWhenRejected`	`true`	A boolean to toggle between the most common rejection policies. • **`true`** is the same as **`rejectedPolicy=CallerRuns`** • **`false`** is the same as **`rejectedPolicy=Abort`**
`executorService`		You can provide a custom **`ExecutorService`** to use, for instance in a managed environment a J2EE container could provide this service so all thread pools is controlled by the J2EE container.
`executorServiceRef`		You can provide a named reference to the custom **`ExecutorService`** from the Camel registry. Keep in mind that reference to the custom executor service cannot be used together with the executor-related options (like **`poolSize`** or **`maxQueueSize`**) **as referenced executor service should be configured already.**

`waitForTaskToComplete`	`IfReplyExpected`	**@deprecated (removed in Camel 2.4):** Option to specify if the caller should wait for the asynchronous task to be complete or not before continuing. The following options are supported: • **`Always`** • **`Never`** • **`IfReplyExpected`** The first two options is self explained. The last will only wait if the message is Request Reply based.

About Rejected Tasks

The **`threads`** DSL uses a thread pool which has a worker queue for tasks. When the worker queue gets full, the task is rejected. You can customize how to react upon this using the **`rejectedPolicy`** and **`callerRunsWhenRejected`** option. The latter is used for easily switch between the two most common and recommended settings. Both let the current caller thread execute the task e.g., it will become synchronous, but also give time for the thread pool to process its current tasks, without adding more tasks - sort of self throttling. This is the default behavior.

If setting **`callerRunsWhenRejected=false`** you use the **`Abort`** policy, which mean the task is rejected, and a **`RejectedExecutionException`** is set on the Exchange, and the Exchange will stop continue being routed, and its **`UnitOfWork`** will be regarded as failed.

The other options **`Discard`** and **`DiscardOldest`** works a bit like **`Abort`**, however they do **not** set any Exception on the Exchange, which mean the Exchange will **not** be regarded as failed, but the Exchange will be successful. When using **`Discard`** and **`DiscardOldest`** then the Exchange will not continue being routed.
Note: there's an issue with these two options in **Camel 2.9** or earlier, that cause the **`UnitOfWork`** not to be triggered, so we discourage you from using these options in those Camel releases. This has been fixed in **Camel 2.10**.

Example: `threads` DSL

Suppose we receive orders on a JMS queue. Some of the orders expect a reply while other do not (either a **`JMSReplyTo`** exists or not). And lets imagine to process this order we need to do some heavy CPU calculation. So how do we avoid the messages that does not expect a reply to block until the entire message is processed? Well we use the **`threads`** DSL to turn the route into multi threading asynchronous routing before the heavy CPU task. Then the messages that does not expect a reply can return beforehand. And the messages that expect a reply, well yeah they have to wait anyway. So this can be accomplished like the route below:{snippet:id=e1|lang=java|url=camel/trunk/tests/camel-itest/src/test/java/org/apache/camel/itest/async/HttpAsyncDslTest.java}
Transactions and threads DSL
Mind that when using transactions its often required that the Exchange is processed entirely in the same thread, as the transaction manager often uses **`ThreadLocal`** to store the intermediate transaction status. For instance Spring Transaction does this. So when using **`threads`** DSL the Exchange that is processed in the asynchronous thread cannot participate in the same transaction as the caller thread.

Note: this does not apply to the **`ProducerTemplate`** Async API as such as the client usually does not participate in a transaction. So you can still use the Camel Client Async API and do asynchronous messaging where the processing of the Exchange is still handled within transaction. It's only the client that submitted the Exchange that does not participate in the same transaction.

See Also

- Asynchronous Processing
- Request Reply
- Request Only
- Blog entry on using async for concurrent file processing
- SEDA
- Direct
- ToAsync for non blocking Request Reply