# Asynchronous Processing

## Asynchronous Processing

### Overview
Supported versions

The information on this page applies for **Camel 2.4** or later.

Before **Camel 2.4** the asynchronous processing is only implemented for JBI where as in **Camel 2.4** we have implemented it in many other areas. See more at Asynchronous Routing Engine.

Camel supports a more complex asynchronous processing model. The asynchronous processors implement the `org.apache.camel.AsyncProcessor` interface which is derived from the more synchronous `org.apache.camel.Processor` interface. There are advantages and disadvantages when using asynchronous processing when compared to using the standard synchronous processing model.

**Advantages**:

- Processing routes that are composed fully of asynchronous processors do not use up threads waiting for processors to complete on blocking calls. This can increase the scalability of your system by reducing the number of threads needed to process the same workload.
- Processing routes can be broken up into SEDA processing stages where different thread pools can process the different stages. This means that your routes can be processed concurrently.

**Disadvantages**:

- Implementing asynchronous processors is more complex than implementing the synchronous versions.

### When to Use

We recommend that processors and components be implemented the more simple synchronous APIs unless you identify a performance of scalability requirement that dictates otherwise. A Processor whose `process()` method blocks for a long time would be good candidates for being converted into an asynchronous processor.

### Interface Details

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

The `AsyncProcessor` defines a single `process()` method which is very similar to it's synchronous `Processor.process()` brethren.

Here are the differences:

- A non-null `AsyncCallback` **MUST** be supplied which will be notified when the exchange processing is completed.
- It **MUST** not throw any exceptions that occurred while processing the exchange. Any such exceptions must be stored on the exchange's `Exception` property.
- It **MUST** know if it will complete the processing synchronously or asynchronously. The method will return `true` if it does complete synchronously, otherwise it returns `false`.
- When the processor has completed processing the exchange, it must call the `callback.done(boolean sync)` method.
- The sync parameter **MUST** match the value returned by the `process()` method.

### Implementing Processors that Use the AsyncProcessor API

All processors, even synchronous processors that do not implement the `AsyncProcessor` interface, can be coerced to implement the `AsyncProcessor` interface. This is usually done when you are implementing a Camel component consumer that supports asynchronous completion of the exchanges that it is pushing through the Camel routes. Consumers are provided a `Processor` object when created. All Processor object can be coerced to a `AsyncProcessor` using the following API:

```
Processor processor = ...
AsyncProcessor asyncProcessor = AsyncProcessorTypeConverter.convert(processor);
```

For a route to be fully asynchronous and reap the benefits to lower Thread usage, it must start with the consumer implementation making use of the asynchronous processing API. If it called the synchronous `process()` method instead, the consumer's thread would be forced to be blocked and in use for the duration that it takes to process the exchange.

It is important to take note that just because you call the asynchronous API, it does not mean that the processing will take place asynchronously. It only allows the possibility that it can be done without tying up the caller's thread. If the processing happens asynchronously is dependent on the configuration of the Camel route.

Normally, the the process call is passed in an inline inner **AsyncCallback** class instance which can reference the exchange object that was declared final. This allows it to finish up any post processing that is needed when the called processor is done processing the exchange.

Example.

```
final Exchange exchange = ...
AsyncProcessor asyncProcessor = ...
asyncProcessor.process(exchange, new AsyncCallback() {
    public void done(boolean sync) {

        if (exchange.isFailed()) {
            ... // do failure processing.. perhaps rollback etc.
        } else {
            ... // processing completed successfully, finish up
                // perhaps commit etc.
        }
    }
});
```

## Asynchronous Route Sequence Scenarios

Now that we have understood the interface contract of the **AsyncProcessor**, and have seen how to make use of it when calling processors, let's looks a what the thread model/sequence scenarios will look like for some sample routes.

The Jetty component's consumers support asynchronous processing through the use of continuations. Suffice to say it can take a HTTP request and pass it to a Camel route for asynchronous processing. If the processing is indeed asynchronous, it uses a Jetty continuation so that the HTTP request is 'parked' and the thread is released. Once the Camel route finishes processing the request, the Jetty component uses the **AsyncCallback** to tell Jetty to 'un-park' the request. Jetty un-parks the request, the HTTP response returned using the result of the exchange processing.

Notice that the jetty continuations feature is only used "If the processing is indeed async". This is why **AsyncProcessor.process()** implementations *must* accurately report if request is completed synchronously or not.

The **jhc** component's producer allows you to make HTTP requests and implement the **AsyncProcessor** interface. A route that uses both the jetty asynchronous consumer and the **jhc** asynchronous producer will be a fully asynchronous route and has some nice attributes that can be seen if we take a look at a sequence diagram of the processing route.

For the route:

```
from("jetty:http://localhost:8080/service")
    .to("jhc:http://localhost/service-impl");
```

The sequence diagram would look something like this:
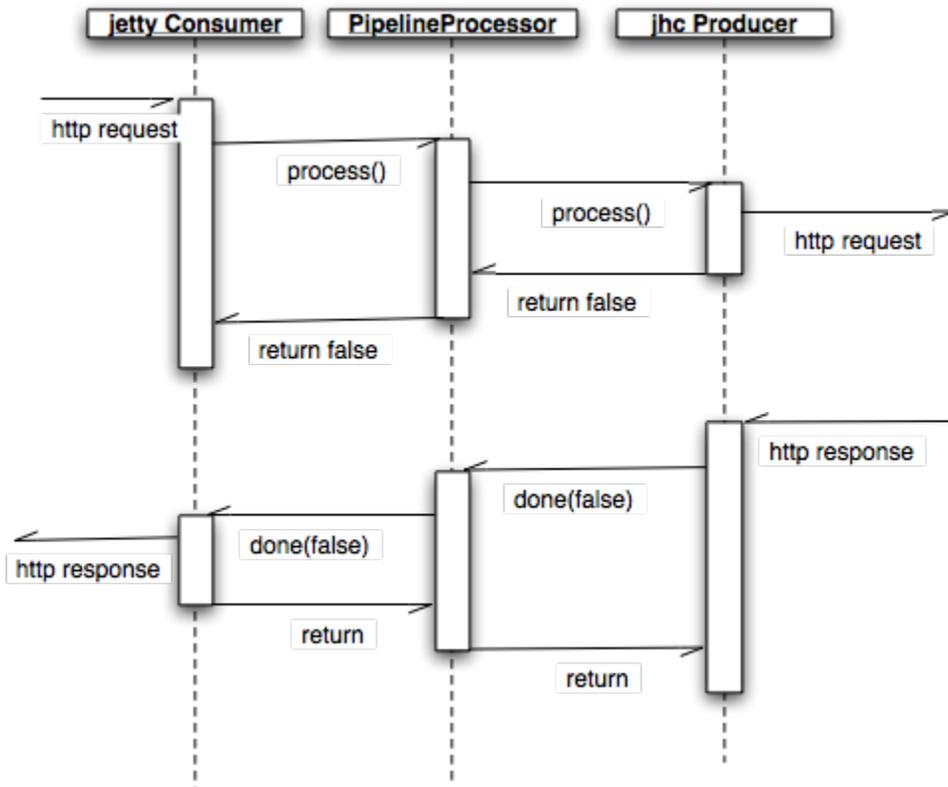
The diagram simplifies things by making it looks like processors implement the `AsyncCallback` interface when in reality the `AsyncCallback` interfaces are inline inner classes, but it illustrates the processing flow and shows how two separate threads are used to complete the processing of the original HTTP request. The first thread is synchronous up until processing hits the `jhc` producer which issues the HTTP request. It then reports that the exchange processing will complete asynchronously using NIO to get the response back. Once the `jhc` component has received a full response it uses `AsyncCallback.done()` method to notify the caller. These callback notifications continue up until it reaches the original Jetty consumer which then un-parks the HTTP request and completes it by providing the response.

## Mixing Synchronous and Asynchronous Processors

It is totally possible and reasonable to mix the use of synchronous and asynchronous processors/components. The pipeline processor is the backbone of a Camel processing route. It glues all the processing steps together. It is implemented as an `AsyncProcessor` and supports interleaving synchronous and asynchronous processors as the processing steps in the pipeline.

Let's say we have two custom asynchronous processors, namely: `MyValidator` and `MyTransformation`. Let's say we want to load file from the data/in directory validate them with the `MyValidator()` processor, transform them into JPA Java objects using `MyTransformation` and then insert them into the database using the JPA component. Let's say that the transformation process takes quite a bit of time and we want to allocate `20` threads to do parallel transformations of the input files. The solution is to make use of the thread processor. The thread is `AsyncProcessor` that forces subsequent processing in asynchronous thread from a thread pool.

The route might look like:

```
from("file:data/in")
   .process(new MyValidator())
   .threads(20)
   .process(new MyTransformation())
   .to("jpa:PurchaseOrder");
```

The sequence diagram would look something like this:
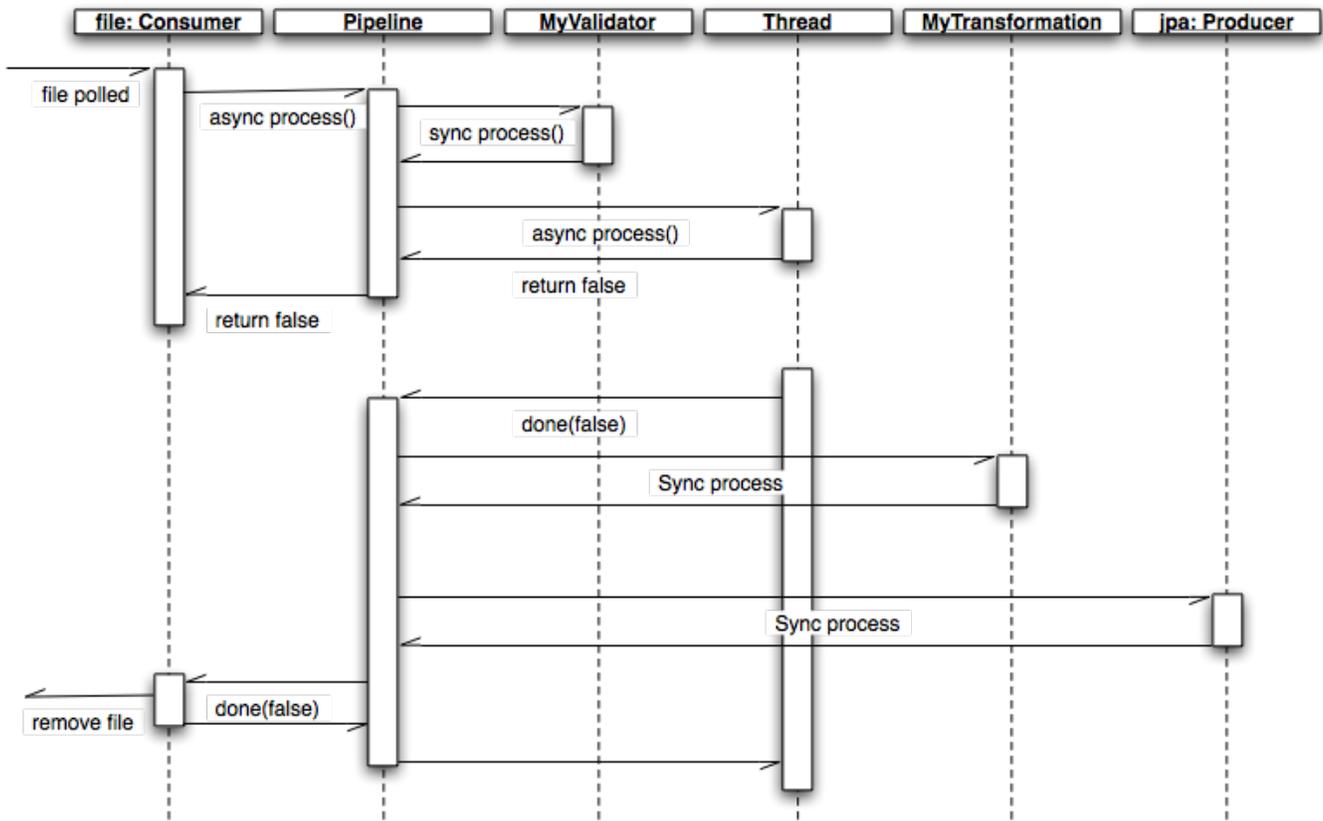
You would actually have multiple threads executing the second part of the thread sequence.

## Staying Synchronous in an `AsyncProcessor`

Generally speaking you get better throughput processing when you process things synchronously. This is due to the fact that starting up an asynchronous thread and doing a context switch to it adds a little bit of of overhead. So it is generally encouraged that `AsyncProcessor`'s do as much work as they can synchronously. When they get to a step that would block for a long time, at that point they should return from the process call and let the caller know that it will be completing the call asynchronously.