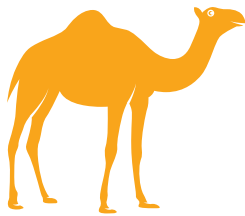


# Implementing Parallel Processing with Apache Camel

Apache Camel offers various components and enterprise integration patterns (EIPs) to achieve concurrency. This article explains the various options available and the best practices to be followed to achieve high scalability when using these EIPs.



Apache Camel provides a number of EIPs (listed below) that allow a main route to divide processing across multiple sub-routes:

- Multicast
- Split
- recipientList
- wireTap

Some of these EIPs provide parallel processing support out-of-the-box, helping to achieve high scalability. Camel ships with default config settings for these EIPs, which can be tuned further to suit one's requirements to get better performance. This article discusses some such useful settings that provide better performance when tuned.

To better illustrate these options let us consider the code snippet given below that uses Camel's Multicast EIP.

We will explore the multiple tuning options available for this EIP and, using sample code snippets, explain how they can be tuned:

```
From ("mainRoute")
    .multicast()
    .aggregationStrategy (new
ResponseAggregator())
    .parallelProcessing()
    .to ("subRoute1", "subRoute2",
subroute3")
```

A message arriving at route 'mainRoute' is being multicast to three different sub-routes which process it in parallel, and once done with processing, the response is aggregated using the aggregation strategy defined in *ResponseAggregator* class. Now let us look at the various fine-tuning options available to achieve high concurrency and scalability out of the above implementations.

## a. Custom thread pool

When using Multicast EIP for parallel processing, Camel uses a default thread pool which has a maximum pool size

of 20, limiting the number of parallel threads that can be spanned to 20:

```
<threadPoolProfile
id="defaultThreadPoolProfile"
defaultProfile="true"

poolSize="10" maxPoolSize="20"
maxQueueSize="1000"

allowCoreThreadTimeOut="false"
rejectedPolicy="CallerRuns"/>
```

With these pool size settings, multicast invocation becomes a major performance bottleneck when processing higher transactions per second (TPS). Also, in cases like the above, where the sub-routes are calls to other components, the pool threads will be majorly in a waiting state and get exhausted quickly. This will result in incoming requests simply waiting for pool threads to free up, leading to an increase in response time and decreased throughput.

It's recommended to use a custom thread pool tuned for the performance needs of each use case rather than using default thread pool settings. All the parallel processing EIPs mentioned above provide a mechanism for passing a custom thread pool. There are two ways to customise a processor's thread pool.

**Approach 1:** Specify a custom thread pool—explicitly create an *ExecutorService* (thread pool) instance and pass it to the *executorService* option. For example:

```
from ("mainRoute")
    .multicast()
```

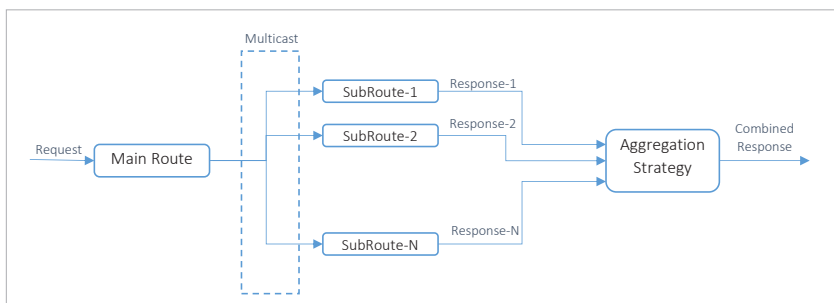


Figure 1: Camel parallel processing

```
.aggregationStrategy (new
ResponseAggregator())
.parallelProcessing()
.executorService (executorService)
.to ("subRoute1", "subRoute2",
subroute3")
```

**Approach 2:** Define a custom thread pool in *camelContext.xml*. You can then reference the custom thread pool using the *executorServiceRef* attribute to look up the thread pool by ID.

```
<threadPool id="customThreadPool"
threadName="customThread"
poolSize="300" maxPoolSize="300"/>


from ("mainRoute")
.multicast()
.aggregationStrategy (new
ResponseAggregator())
.parallelProcessing()
.executorServiceRef
(customThreadPool)
.to ("subRoute1", "subRoute2",
subroute3")
```

Approach 2 is preferred since the thread pool configuration goes into the configuration file (*camelContext.xml*) and can be modified without changing any code.

**A few points about the configuration for the custom thread pool:** Define a unique pool for each parallel processing flow and tune the pool based on the requirements of the route. A few factors to consider – the maximum load expected to be handled by the main route, the number of sub-routes to be processed by the thread pool, and the ratio of processing time vs wait time expected when the threads are executing the sub-routes. Determine the number of threads needed and use the same value for both *poolSize* and *maxPoolSize*. Quoting from Javadoc on the behaviour of pool sizes: “If there are more than *corePoolSize* but less than *maximumPoolSize* threads running, a new thread will be created only if the queue is full.”

## b. Streaming

When using the parallel processing EIPs, we specify an *AggregationStrategy* that aggregates/combines the responses from parallel sub-routes into one combined response. However, by default, the responses are aggregated in the same order in which the parallel sub-routes are invoked. This default behaviour causes aggregation tasks to spend lot of CPU in the polling mechanism. By enabling streaming we can reduce this CPU usage and provide a better performance. With streaming, the responses will be processed as and when they are received rather than in the order of multicast route invocation.


 **Note:** Streaming should be applied only if the *AggregationStrategy* does not depend on the order of responses from the sub-routes. An example is:

```
from ("mainRoute")
.multicast()
.aggregationStrategy (new
ResponseAggregator())
.parallelProcessing()
.streaming()
.executorService(executorService)
.to ("subRoute1", "subRoute2",
subroute3")
```

## c. Parallel aggregation

*AggregationStrategy* combines the responses from parallel sub-routes into one final response. This is done by invoking the *aggregate()* method in the *AggregationStrategy* class for each response received from the sub-route.

By default, Camel synchronises the call to the aggregate method. If parallel aggregation is enabled, then the aggregate method on *AggregationStrategy* can be called concurrently. This can be used to achieve higher performance when the *AggregationStrategy* is implemented as thread safe.

 **Note:** Enabling parallel aggregation would require the *AggregationStrategy* to be implemented as thread safe. An example is:

```
from ("mainRoute")
.multicast()
.aggregationStrategy(new
ResponseAggregator())
.parallelAggregate()
.parallelProcessing()
.streaming()
.executorService (executorService)
.to ("subRoute1", "subRoute2",
subroute3")
```

END 

## Disclaimer

The views expressed in this paper solely belong to the author, and does not reflect the views of their employer (Sabre).

## References

- [1] <https://camel.apache.org/components/latest/eips/enterprise-integration-patterns.html>
- [2] <https://camel.apache.org/components/latest/eips/multicast-eip.html>
- [3] <https://camel.apache.org/manual/latest/threading-model.html>
- [4] [https://access.redhat.com/documentation/en-us/red\\_hat\\_fuse/7.3/html/apache\\_camel\\_development\\_guide/basicprinciples#BasicPrinciples-Thread](https://access.redhat.com/documentation/en-us/red_hat_fuse/7.3/html/apache_camel_development_guide/basicprinciples#BasicPrinciples-Thread)
- [5] [https://access.redhat.com/documentation/en-us/red\\_hat\\_fuse/7.3/html/apache\\_camel\\_development\\_guide/msgrount#MsgRout-Multicast](https://access.redhat.com/documentation/en-us/red_hat_fuse/7.3/html/apache_camel_development_guide/msgrount#MsgRout-Multicast)

## By: Lokesh Chenta

The author is a principal software architect in the lodging, ground and sea (LGS) team at Sabre's Bengaluru Global Capability Center. He has more than 14 years of experience in designing and developing scalable, enterprise applications.