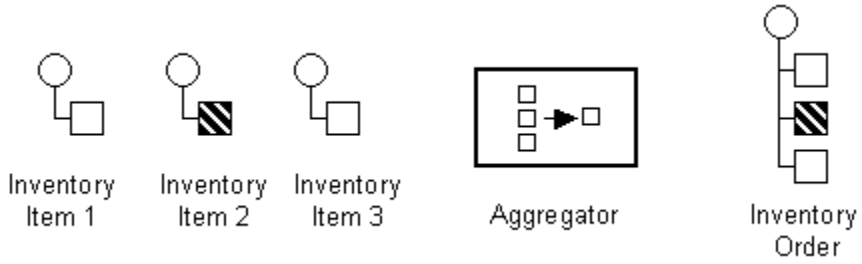
Aggregator2

Aggregator

This applies for Camel version 2.3 or newer. If you use an older version then use this Aggregator link instead.

The Aggregator from the EIP patterns allows you to combine a number of messages together into a single message.



A correlation Expression is used to determine the messages which should be aggregated together. If you want to aggregate all messages into a single message, just use a constant expression. An **`AggregationStrategy`** is used to combine all the message exchanges for a single correlation key into a single message exchange.

Aggregator options

The aggregator supports the following options:

confluenceTableSmall

Option	Default	Description
`correlationExpression`		Mandatory Expression which evaluates the correlation key to use for aggregation. The Exchange which has the same correlation key is aggregated together. If the correlation key could not be evaluated an Exception is thrown. You can disable this by using the **`ignoreBadCorrelationKeys`** option.
`aggregationStrategy`		Mandatory **`AggregationStrategy`** which is used to *merge* the incoming Exchange with the existing already merged exchanges. At first call the **`oldExchange`** parameter is **`null`**. On subsequent invocations the **`oldExchange`** contains the merged exchanges and **`newExchange`** is of course the new incoming Exchange. From **Camel 2.9.2** onwards the strategy can also be a **`TimeoutAwareAggregationStrategy`** implementation, supporting the timeout callback, see further below for more details. From **Camel 2.16**: the strategy can also be a **`PreCompletionAwareAggregationStrategy`** implementation which then runs the completion check in pre-completion mode. See further below for more details.
`strategyRef`		A reference to lookup the **`AggregationStrategy`** in the Registry. From **Camel 2.12** onwards you can also use a POJO as the **`AggregationStrategy`**, see further below for details.
`strategyMethodName`		**Camel 2.12:** This option can be used to explicit declare the method name to use, when using POJOs as the **`AggregationStrategy`**. See further below for more details.
`strategyMethodAllowNull`	`false`	**Camel 2.12:** If this option is **`false`** then the aggregate method is not used for the very first aggregation. If this option is **`true`** then **`null`** values is used as the **`oldExchange`** (at the very first aggregation), when using POJOs as the **`AggregationStrategy`**. See further below for more details.
`completionSize`		Number of messages aggregated before the aggregation is complete. This option can be set as either a fixed value or using an Expression which allows you to evaluate a size dynamically - will use **`Integer`** as result. If both are set Camel will fallback to use the fixed value if the Expression result was **`null`** or **`0`**.
`completionTimeout`		Time in millis that an aggregated exchange should be inactive before its complete. This option can be set as either a fixed value or using an Expression which allows you to evaluate a timeout dynamically - will use **`Long`** as result. If both are set Camel will fallback to use the fixed value if the Expression result was **`null`** or **`0`**. You cannot use this option together with **`completionInterval`**, only one of the two can be used.
`completionInterval`		A repeating period in millis by which the aggregator will complete all current aggregated exchanges. Camel has a background task which is triggered every period. You cannot use this option together with **`completionTimeout`**, only one of them can be used.

`completionPredicate`		A Predicate to indicate when an aggregated exchange is complete. From **Camel 2.15**: if this is not specified and the **`AggregationStrategy`** object implements Predicate, the **`aggregationStrategy`** object will be used as the **`completionPredicate`**.
`completionFromBatchConsumer`	`false`	This option is if the exchanges are coming from a Batch Consumer. Then when enabled the Aggregator2 will use the batch size determined by the Batch Consumer in the message header **`CamelBatchSize`**. See more details at Batch Consumer. This can be used to aggregate all files consumed from a File endpoint in that given poll.
`forceCompletionOnStop`	`false`	**Camel 2.9** Indicates to complete all current aggregated exchanges when the context is stopped
`completeAllOnStop`	`false`	**Camel 2.16:** Indicates to wait to complete all current and partial (pending) aggregated exchanges when the context is stopped. This also means that we will wait for all pending exchanges which are stored in the aggregation repository to complete so the repository is empty before we can stop. You may want to enable this when using the memory based aggregation repository that is memory based only, and do not store data on disk. When this option is enabled, then the aggregator is waiting to complete all those exchanges before its stopped, when stopping CamelContext or the route using it.
`eagerCheckCompletion`	`false`	Whether or not to eager check for completion when a new incoming Exchange has been received. This option influences the behavior of the **`completionPredicate`** option as the Exchange being passed in changes accordingly. When **`false`** the Exchange passed in the Predicate is the *aggregated* Exchange which means any information you may store on the aggregated Exchange from the **`AggregationStrategy`** is available for the Predicate. When `true` the Exchange passed in the Predicate is the *incoming* Exchange, which means you can access data from the incoming Exchange.
`groupExchanges`	`false`	If enabled then Camel will group all aggregated Exchanges into a single combined **`org.apache.camel.impl.GroupedExchange`** holder class that holds all the aggregated Exchanges. And as a result only one Exchange is being sent out from the aggregator. Can be used to combine many incoming Exchanges into a single output Exchange without coding a custom **`AggregationStrategy`** yourself. **Note:** this option does **not** support persistent repository with the aggregator. See further below for an example and more details.
`ignoreInvalidCorrelationKeys`	`false`	Whether or not to ignore correlation keys which could not be evaluated to a value. By default Camel will throw an Exception, but you can enable this option and ignore the situation instead.
`closeCorrelationKeyOnCompletion`		Whether or not too *late* Exchanges should be accepted or not. You can enable this to indicate that if a correlation key has already been completed, then any new exchanges with the same correlation key be denied. Camel will then throw a **`ClosedCorrelationKeyException`** exception. When using this option you pass in a **`integer`** which is a number for a LRUCache which keeps that last X number of closed correlation keys. You can pass in 0 or a negative value to indicate a unbounded cache. By passing in a number you are ensured that cache won't grow too big if you use a log of different correlation keys.
`discardOnCompletionTimeout`	`false`	**Camel 2.5:** Whether or not exchanges which complete due to a timeout should be discarded. If enabled then when a timeout occurs the aggregated message will **not** be sent out but dropped (discarded).
`aggregationRepository`		Allows you to plugin you own implementation of **`org.apache.camel.spi.AggregationRepository`** which keeps track of the current inflight aggregated exchanges. Camel uses by default a memory based implementation.
`aggregationRepositoryRef`		Reference to lookup a **`aggregationRepository`** in the Registry.
`parallelProcessing`	`false`	When aggregated are completed they are being send out of the aggregator. This option indicates whether or not Camel should use a thread pool with multiple threads for concurrency. If no custom thread pool has been specified then Camel creates a default pool with 10 concurrent threads.
`executorService`		If using **`parallelProcessing`** you can specify a custom thread pool to be used. In fact also if you are not using **`parallelProcessing`** this custom thread pool is used to send out aggregated exchanges as well.
`executorServiceRef`		Reference to lookup a **`executorService`** in the Registry
`timeoutCheckerExecutorService`		**Camel 2.9:** If using either of the **`completionTimeout`**, **`completionTimeoutExpression`**, or **`completionInterval`** options a background thread is created to check for the completion for every aggregator. Set this option to provide a custom thread pool to be used rather than creating a new thread for every aggregator.
`timeoutCheckerExecutorServiceRef`		**Camel 2.9:** Reference to lookup a **`timeoutCheckerExecutorService`** in the Registry

`optimisticLocking`	`false`	**Camel 2.11:** Turns on using optimistic locking, which requires the **`aggregationRepository`** being used, is supporting this by implementing the **`org.apache.camel.spi.OptimisticLockingAggregationRepository`** interface.
`optimisticLockRetryPolicy`		**Camel 2.11.1:** Allows to configure retry settings when using optimistic locking.

Exchange Properties

The following properties are set on each aggregated Exchange:

confluenceTableSmall

Header	Type	Description
`CamelAggregatedSize`	`int`	The total number of Exchanges aggregated into this combined Exchange.
`CamelAggregatedCompletedBy`	`String`	Indicator how the aggregation was completed as a value of either: **`predicate`**, **`size`**, **`strategy`**, **`consumer`**, **`timeout`**, **`forceCompletion`** or **`interval`**.

About AggregationStrategy

The **`AggregationStrategy`** is used for aggregating the old (lookup by its correlation id) and the new exchanges together into a single exchange. Possible implementations include performing some kind of combining or delta processing, such as adding line items together into an invoice or just using the newest exchange and removing old exchanges such as for state tracking or market data prices; where old values are of little use.

Notice the aggregation strategy is a mandatory option and must be provided to the aggregator.

Here are a few example **`AggregationStrategy`** implementations that should help you create your own custom strategy.
class ArrayListAggregationStrategy implements AggregationStrategy { public Exchange aggregate(Exchange oldExchange, Exchange newExchange) { Object newBody = newExchange.getIn().getBody(); ArrayList