Aggregator

Aggregator

This applies for Camel version 2.2 or older. If you use a newer version then the Aggregator has been rewritten from Camel 2.3 on and you should use this Aggregator2 link instead.

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

blocked URL

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. The default strategy just chooses the latest message; so its ideal for throttling messages.

For example, imagine a stock market data system; you are receiving 30,000 messages per second; you may want to throttle down the updates as, say, a GUI cannot cope with such massive update rates. So you may want to aggregate these messages together so that within a window (defined by a maximum number of messages or a timeout), messages for the same stock are aggregated together; by just choosing the latest message and discarding the older prices. (You could apply a delta processing algorithm if you prefer to capture some of the history).

Using the aggregator correctly

Torsten Mielke wrote a nice blog entry with his thoughts and experience on using the aggreagator. Its a well worth read.

AggregationStrategy changed in Camel 2.0

In Camel 2.0 the **`AggregationStrategy`** callback have been changed to also be invoked on the very first Exchange.

On the first invocation of the **`aggregate`** method the **`oldExchange`** parameter is **`null`**. The reason is that we have not aggregated anything yet.
So its only the **`newExchange`** that has a value. Usually you just return the **`newExchange`** in this situation. But you still have the power to decide what to do, for example you can do some alternation on the exchange or remove some headers. And a more common use case is for instance to count some values from the body payload. That could be to sum up a total amount etc.

BatchTimeout and CompletionPredicate

You cannot use both **`batchTimeout`** and **`completionPredicate`** to trigger a completion based on either on reaching its goal first. The batch timeout will always trigger first, at that given interval.

Using the Fluent Builders

The following example shows how to aggregate messages so that only the latest message for a specific value of the **`cheese`** header are sent.

Error formatting macro: snippet: java.lang.NullPointerException

If you were using JMS then you may wish to use the **`JMSDestination`** header as the correlation key; or some custom header for the stock symbol (using the above stock market example).

```
from("activemq:someReallyFastTopic")
  .aggregator(header("JMSDestination"))
  .to("activemq:someSlowTopicForGuis");
```

You can of course use many different Expression languages such as XPath, XQuery, SQL or various Scripting Languages.
Here is an example using **XPath**:

```
//aggregate based on the message content using an XPath expression
//example assumes an XML document starting with <stockQuote symbol='...'>
//aggregate messages based on their symbol attribute within the <stockQuote> element
from("seda:start").aggregate().xpath("/stockQuote/@symbol", String.class).batchSize(5).to("mock:result");

//this example will aggregate all messages starting with <stockQuote symbol='APACHE'> into
//one exchange and all the other messages (different symbol or different root element) into another exchange.
from("seda:start").aggregate().xpath("name(/stockQuote[@symbol='APACHE'])", String.class).batchSize(5).to("mock:
result");
```

For further examples of this pattern in use you could look at the junit test case

Using the Spring XML Extensions

The **correlationExpression** element is in Camel 2.0. For earlier versions of Camel you will need to specify your expression without the enclosing **correlationExpression** element.

```
<aggregator>
  <simple>header.cheese</simple>
  <to uri="mock:result"/>
</aggregator>
```

The following example shows how to create a simple aggregator using the XML notation; using an Expression for the correlation value used to aggregate messages together.

Error formatting macro: snippet: java.lang.NullPointerException

You can specify your own **AggregationStrategy** if you prefer as shown in the following example

Error formatting macro: snippet: java.lang.NullPointerException

Notice how the **strategyRef** attribute is used on the **<aggregator>** element to refer to the custom strategy in Spring.

Exchange Properties

The following properties is set on each Exchange that are aggregated:

header	type	description
`org.apache.camel.Exchange.AggregatedCount`	int	Camel 1.x: The total number of Exchanges aggregated in this combined Exchange.
`CamelAggregatedSize`	int	Camel 2.0: The total number of Exchanges aggregated into this combined Exchange.
`CamelAggregatedIndex`	int	Camel 2.0: The current index of this Exchange in the batch.

Batch options

The aggregator supports the following batch options:

Option	Default	Description
`batchSize`	`100`	The **in** batch size. This is the number of incoming exchanges that is processed by the aggregator and when this threshold is reached the batch is completed and send. **Camel 1.6.2/2.0:** You can disable the batch size so the Aggregator is only triggered by timeout by setting the **batchSize** to 0 (or negative). In **Camel 1.6.1** or older you can set the **batchSize** to a very large number to archive the same.
`outBatchSize`	`0`	**Camel 1.5:** The **out** batch size. This is the number of exchanges currently aggregated in the **AggregationCollection**. When this threshold is reached the batch is completed and send. By default this option is disabled. The difference to the **batchSize** options is that this is for outgoing, so setting this size to e.g. 50 ensures that this batch will at maximum contain 50 exchanges when its sent.
`batchTimeout`	`1000L`	Timeout in millis. How long should the aggregator wait before its completed and sends whatever it has currently aggregated.
`groupExchanges`	`false`	**Camel 2.0**: 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.
`batchConsumer`	`false`	**Camel 2.0**: This option is if the exchanges is coming from a Batch Consumer. Then when enabled the Aggregator 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.
`completionPredicate`	`null`	Allows you to use a Predicate to signal when an aggregation is complete. See **warning** in top of this page.

AggregationCollection and AggregationStrategy

This aggregator uses a **AggregationCollection** to store the exchanges that is currently aggregated. The **AggregationCollection** uses a correlation Expression and an **AggregationStrategy**.

- The correlation Expression is used to correlate the incoming exchanges. The default implementation will group messages based on the correlation expression. Other implementations could for instance just add all exchanges as a batch.

- The strategy is used for aggregate 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.

Camel provides these implementations:

- **`DefaultAggregationCollection`**
- **`PredicateAggregationCollection`**
- **`UseLatestAggregationStrategy`**

Examples

Default example

By default Camel uses **`DefaultAggregationCollection`** and **`UseLatestAggregationStrategy`**, so this simple example will just keep the latest received exchange for the given correlation Expression:

Error formatting macro: snippet: java.lang.NullPointerException

Using `PredicateAggregationCollection`

The **`PredicateAggregationCollection`** is an extension to **`DefaultAggregationCollection`** that uses a Predicate as well to determine the completion. For instance the Predicate can test for a special header value, a number of maximum aggregated so far etc. To use this the routing is a bit more complex as we need to create our **`AggregationCollection`** object as follows:

Error formatting macro: snippet: java.lang.NullPointerException

In this sample we use the predicate that we want at most 3 exchanges aggregated by the same correlation id, this is defined as:

```
header(Exchange.AGGREGATED_COUNT).isEqualTo(3)
```

Using this the aggregator will complete if we receive 3 exchanges with the same correlation id or when the specified timeout of 500 msecs has elapsed (whichever criteria is met first).

Using Custom Aggregation Strategy

In this example we will aggregate incoming bids and want to aggregate the highest bid. So we provide our own strategy where we implement the code logic:

Error formatting macro: snippet: java.lang.NullPointerException

Then we setup the routing as follows:

Error formatting macro: snippet: java.lang.NullPointerException

And since this is based on an unit test we show the test code that send the bids and what is expected as the **`winners`**:

Error formatting macro: snippet: java.lang.NullPointerException

Using Custom Aggregation Collection

In this example we will aggregate incoming bids and want to aggregate the bids in reverse order (this is just an example). So we provide our own collection where we implement the code logic:

Error formatting macro: snippet: java.lang.NullPointerException

Then we setup the routing as follows:

Error formatting macro: snippet: java.lang.NullPointerException

And since this is based on an unit test we show the test code that send the bids and what is expected as the expected reverse order:

Error formatting macro: snippet: java.lang.NullPointerException

Custom aggregation collection in Spring DSL

You can also specify a custom aggregation collection in the Spring DSL. Here is an example for Camel 2.0

Error formatting macro: snippet: java.lang.NullPointerException

In Camel 1.5.1 you will need to specify the aggregator as:

```
<aggregator batchTimeout="500" collectionRef="aggregatorCollection">
  <expression/>
```

```
  <to uri="mock:result"/>
</aggregator>
```

Using Grouped Exchanges

Available as of Camel 2.0

You can enable grouped exchanges to combine all aggregated exchanges into a single **`org.apache.camel.impl.GroupedExchange`** holder class that contains all the individual aggregated exchanges. This allows you to process a single Exchange containing all the aggregated exchange. Lets start with how to configure this in the router:

Error formatting macro: snippet: java.lang.NullPointerException

And the next part is part of an unit code that demonstrates this feature as we send in 5 exchanges each with a different value in the body. And we will only get 1 exchange out of the aggregator, but we can access all the individual aggregated exchanges from the List which we can extract as a property from the Exchange using the key **`Exchange.GROUPED_EXCHANGE`**.

Error formatting macro: snippet: java.lang.NullPointerException

Using Batch Consumer

Available as of Camel 2.0

The Aggregator can work together with the Batch Consumer to aggregate the total number of messages that the Batch Consumer have reported. This allows you for instance to aggregate all files polled using the File consumer.

For example:

```
from("file://inbox")
   .aggregate(xpath("//order/@customerId"), new AggregateCustomerOrderStrategy()).batchConsumer().batchTimeout
(60000).to("bean:processOrder");
```

When using **`batchConsumer`** Camel will automatic adjust the **`batchSize`** according to reported by the Batch Consumer in this case the file consumer. So if we poll in 7 files then the aggregator will aggregate all 7 files before it completes. As the timeout is still in play we set it to 60 seconds.

Using This Pattern

If you would like to use this EIP Pattern then please read the Getting Started, you may also find the Architecture useful particularly the description of Endpoint and URIs. Then you could try out some of the Examples first before trying this pattern out.

See also

- The Loan Broker Example which uses an aggregator
- Blog post by Torsten Mielke about using the aggregator correctly.