Aggregator

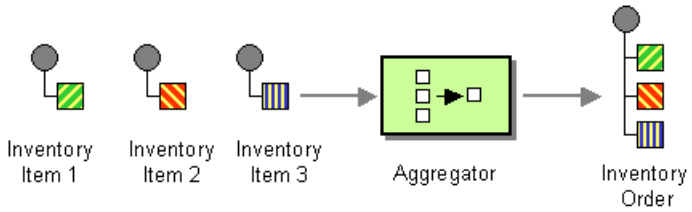
Overview

The *aggregator* pattern enables you to combine a batch of related messages into a single message. To control the aggregator's behavior, Mediation Router allows you to specify the properties described in *Enterprise Integration Patterns*, as follows:

- *Correlation expression*—determines which messages should be aggregated together. The correlation expression is evaluated on each incoming message to produce a *correlation key*. Incoming messages with the same correlation key are then grouped into the same batch. For example, if you want to aggregate *all* incoming messages into a single message, you could use a constant expression.
- *Completeness condition*—determines when a batch of messages is complete. You can specify this either as a simple size limit or, more generally, you can specify a predicate condition that flags when the batch is complete.
- *Aggregation algorithm*—combines the message exchanges for a single correlation key into a single message exchange. The default strategy simply chooses the latest message, which makes it ideal for throttling message flows.

For example, consider a stock market data system that receives 30,000 messages per second. You might want to throttle down the message flow if, say, your GUI tool cannot cope with such a massive update rate. The incoming stock quotes could be aggregated together simply by choosing the latest quote and discarding the older prices. (You could apply a delta processing algorithm, if you prefer to capture some of the history.)

Figure 19. Aggregator Pattern



Simple aggregator

You can define a simple aggregator by calling the `aggregator()` DSL command with a correlation expression as its sole argument (default limits are applied to the batch size—see Specifying the batch size on page 62). The following example shows how to aggregate stock quotes, so that only the latest quote is propagated for the symbol contained in the `StockSymbol` header:

```
from("direct:start").aggregator(header("StockSym
bol")).to("mock:result");
```

The following example shows how to configure the same route using XML configuration:

```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregator>
      <simple>header.StockSymbol</simple>
      <to uri="mock:result"/>
    </aggregator>
  </route>
</camelContext>
```

Specifying the batch size

Normally, you would also specify how many messages should be collected (the *batch size*) before the aggregate message gets propagated to the target endpoint. Mediation Router provides several different settings for controlling the batch size, as follows:

- *Batch size*—specifies an upper limit to the number of messages in a batch (default is 100). For example, the following Java DSL route sets an upper limit of 10 message in a batch:

```
from("direct:start").aggregator(header("StockSymbol")).batch
Size(10).to("mock:result");
```

The following example shows how to configure the same route using XML:

```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregator batchSize="10">
      <simple>header.StockSymbol</simple>
      <to uri="mock:result"/>
    </aggregator>
  </route>
</camelContext>
```

- *Batch timeout*—specifies a time interval, in units of milliseconds, during which messages are collected (default is 1000 ms). If no messages are received during a given time interval, no aggregate message will be propagated. For example, the following Java DSL route aggregates the messages that arrive during each ten second window:

```
from("direct:start").aggregator(header("StockSymbol")).batch
Timeout(10000).to("mock:result");
```

The following example shows how to configure the same route using XML:

```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregator batchTimeout="10000">
      <simple>header.StockSymbol</simple>
      <to uri="mock:result"/>
    </aggregator>
  </route>
</camelContext>
```

- *Completed predicate*—specifies an arbitrary predicate expression that determines when the current batch is complete. If the predicate resolves to `true`, the current message becomes the last message of the batch. For example, if you want to terminate a batch of stock quotes every time you receive an `ALERT` message (as indicated by the value of a `MsgType` header), you could define a route like the following:

```
from("direct:start").aggregator(header("StockSymbol")).
    completedPredicate(header("Ms
gType").isEqualTo("ALERT")).to("mock:result");
```

The following example shows how to configure the same route using XML:

```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregator>
      <simple>header.StockSymbol</simple>
      <completedPredicate>
          <simple>header.MsgType = 'ALERT'</simple>
      </completedPredicate>
      <to uri="mock:result"/>
    </aggregator>
  </route>
</camelContext>
```

You can also combine batch limiting mechanisms, in which case a batch is completed whenever the first of the limits is reached. For example, to specify all three limits simultaneously:

```
from("direct:start").aggregator(header("StockSymbol")).
    batchSize(10).
    batchTimeout(10000).
   completedPredicate(header("MsgType").isEqualTo("ALERT")).

    to("mock:result");
```

Custom aggregation strategy

The default aggregation strategy is to select the most recent message in a batch, discarding all others. If you want to apply a different aggregation strategy, you can implement a custom version of the `org.apache.camel.processor.aggregate.AggregationStrategy` interface and pass it as the second argument to the `aggregator`() DSL command. For example, to aggregate messages using the custom strategy class, `MyAggregationStrategy`, you could define a route like the following:

```
from("direct:start").aggregator(header("StockSymbol"), new
MyAggregationStrategy()).to("mock:result");
```

The following code implements a custom aggregation strategy, `MyAggregationStrategy`, that concatenates all of the batch messages into a single, large message:

```
// Java
package com.my_package_name

import org.apache.camel.processor.aggregate.Aggregation
Strategy;
import org.apache.camel.Exchange;

public class MyAggregationStrategy implements Aggregation
Strategy {
    public Exchange aggregate(Exchange oldExchange, Exchange
 newExchange) {
        String oldBody = oldExchange.getIn().get
Body(String.class);
        String newBody = newExchange.getIn().get
Body(String.class);
        String concatBody = oldBody.concat(newBody);
        // Set the body equal to a concatenation of old and
new.
        oldExchange.getIn().setBody(concatBody);
        // Ignore the message headers!
       // (in a real application, you would probably want to
 do
      //  something more sophisticated with the header data).

        return oldExchange;
    }
}
```

You can also configure a route with a custom aggregation strategy in XML, as follows:

```
<camelContext id="camel" xmlns="http://act
ivemq.apache.org/camel/schema/spring">
  <route>
    <from uri="direct:start"/>
    <aggregator strategyRef="aggregatorStrategy">
      <simple>header.StockSymbol</simple>
      <to uri="mock:result"/>
    </aggregator>
  </route>
</camelContext>

<bean id="aggregatorStrategy" class="com.my_package_name.MyAg
gregationStrategy"/>
```