[Camel](http://camel.apache.org/) [Apache](http://www.apache.org)

[Apache Camel](http://camel.apache.org/index.html) > [Documentation](http://camel.apache.org/documentation.html) > [Enterprise Integration Patterns](http://camel.apache.org/enterprise-integration-patterns.html) > [Splitter](http://camel.apache.org/splitter.html)

[Download](http://camel.apache.org/download.html) | [JavaDoc](http://camel.apache.org/javadoc.html) | [Source](http://camel.apache.org/source.html) | [Forums](http://camel.apache.org/discussion-forums.html) | [Support](http://camel.apache.org/support.html)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Splitter**  The [Splitter](http://www.enterpriseintegrationpatterns.com/Sequencer.html) from the [EIP patterns](http://camel.apache.org/enterprise-integration-patterns.html) allows you split a message into a number of pieces and process them individually  http://www.enterpriseintegrationpatterns.com/img/Sequencer.gif  You need to specify a Splitter as split(). In earlier versions of Camel, you need to use splitter().  **Options**   |  |  |  | | --- | --- | --- | | **Name** | **Default Value** | **Description** | | strategyRef |  | Refers to an [AggregationStrategy](http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/processor/aggregate/AggregationStrategy.html) to be used to assemble the replies from the sub-messages, into a single outgoing message from the [Splitter](http://camel.apache.org/splitter.html). See the defaults described below in [*What the Splitter returns*](http://camel.apache.org/splitter.html#Splitter-WhattheSplitterreturns). From **Camel 2.12** onwards you can also use a POJO as the AggregationStrategy, see the [Aggregate](http://camel.apache.org/aggregator2.html) page for more details. If an exception is thrown from the aggregate method in the AggregationStrategy, then by default, that exception is not handled by the error handler. The error handler can be enabled to react if enabling the shareUnitOfWork option. | | strategyMethodName |  | **Camel 2.12:** This option can be used to explicit declare the method name to use, when using POJOs as the AggregationStrategy. See the [Aggregate](http://camel.apache.org/aggregator2.html) page for more details. | | strategyMethodAllowNull | false | **Camel 2.12:** If this option is false then the aggregate method is not used for the very first splitted message. If this option is true then null values is used as the oldExchange (for the very first message splitted), when using POJOs as the AggregationStrategy. See the [Aggregate](http://camel.apache.org/aggregator2.html) page for more details. | | parallelProcessing | false | If enabled then processing the sub-messages occurs concurrently. Note the caller thread will still wait until all sub-messages has been fully processed, before it continues. |  | | parallelAggregate | false | **Camel 2.14:** If enabled then the aggregate method on AggregationStrategy can be called concurrently. Notice that this would require the implementation of AggregationStrategy to be implemented as thread-safe. By default this is false meaning that Camel synchronizes the call to the aggregate method. Though in some use-cases this can be used to archive higher performance when the AggregationStrategy is implemented as thread-safe. |  | | executorServiceRef |  | Refers to a custom [Thread Pool](http://camel.apache.org/threading-model.html) to be used for parallel processing. Notice if you set this option, then parallel processing is automatically implied, and you do not have to enable that option as well. |  | | stopOnException | false | **Camel 2.2:** Whether or not to stop continue processing immediately when an exception occurred. If disable, then Camel continue splitting and process the sub-messages regardless if one of them failed. You can deal with exceptions in the [AggregationStrategy](http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/processor/aggregate/AggregationStrategy.html) class where you have full control how to handle that. |  | | streaming | false | If enabled then Camel will split in a streaming fashion, which means it will split the input message in chunks. This reduces the memory overhead. For example if you split big messages its recommended to enable streaming. If streaming is enabled then the sub-message replies will be aggregated out-of-order, eg in the order they come back. If disabled, Camel will process sub-message replies in the same order as they where splitted. |  | | timeout |  | **Camel 2.5:** Sets a total timeout specified in millis. If the [Recipient List](http://camel.apache.org/recipient-list.html) hasn't been able to split and process all replies within the given timeframe, then the timeout triggers and the [Splitter](http://camel.apache.org/splitter.html) breaks out and continues. Notice if you provide a [TimeoutAwareAggregationStrategy](http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/processor/aggregate/TimeoutAwareAggregationStrategy.html) then the timeout method is invoked before breaking out. If the timeout is reached with running tasks still remaining, certain tasks for which it is difficult for Camel to shut down in a graceful manner may continue to run. So use this option with a bit of care. We may be able to improve this functionality in future Camel releases. |  | | onPrepareRef |  | **Camel 2.8:** Refers to a custom [Processor](http://camel.apache.org/processor.html) to prepare the sub-message of the [Exchange](http://camel.apache.org/exchange.html), before its processed. This allows you to do any custom logic, such as deep-cloning the message payload if that's needed etc. |  | | shareUnitOfWork | false | **Camel 2.8:** Whether the unit of work should be shared. See further below for more details. |  |   **Exchange properties**  The following properties are set on each Exchange that are split:   |  |  |  | | --- | --- | --- | | **property** | **type** | **description** | | CamelSplitIndex | int | A split counter that increases for each Exchange being split. The counter starts from 0. | | CamelSplitSize | int | The total number of Exchanges that was splitted. This header is not applied for stream based splitting. From **Camel 2.9** onwards this header is also set in stream based splitting, but only on the completed Exchange. | | CamelSplitComplete | boolean | **Camel 2.4:** Whether or not this Exchange is the last. |   **Examples**  The following example shows how to take a request from the **direct:a** endpoint the split it into pieces using an [Expression](http://camel.apache.org/expression.html), then forward each piece to **direct:b**  **Using the** [**Fluent Builders**](http://camel.apache.org/fluent-builders.html)   |  | | --- | | RouteBuilder builder = new RouteBuilder() {      public void configure() {          errorHandler(deadLetterChannel("mock:error"));            from("direct:a")              .split(body(String.class).tokenize("\n"))                  .to("direct:b");      }  }; |   The splitter can use any [Expression](http://camel.apache.org/expression.html) language so you could use any of the [Languages Supported](http://camel.apache.org/languages-supported.html) such as [XPath](http://camel.apache.org/xpath.html), [XQuery](http://camel.apache.org/xquery.html), [SQL](http://camel.apache.org/sql.html) or one of the [Scripting Languages](http://camel.apache.org/scripting-languages.html) to perform the split. e.g.   |  | | --- | | from("activemq:my.queue").split(xpath("//foo/bar")).convertBodyTo(String.class).to("[file://some/directory"](file:///\\some\directory%22)) |   **Using the** [**Spring XML Extensions**](http://camel.apache.org/spring-xml-extensions.html)   |  | | --- | | <camelContext errorHandlerRef="errorHandler" xmlns="[http://camel.apache.org/schema/spring"](http://camel.apache.org/schema/spring%22)>      <route>          <from uri="direct:a"/>          <split>              <xpath>/invoice/lineItems</xpath>              <to uri="direct:b"/>          </split>      </route>  </camelContext> |   For further examples of this pattern in use you could look at one of the [junit test case](http://svn.apache.org/viewvc/camel/trunk/camel-core/src/test/java/org/apache/camel/processor/SplitterTest.java?view=markup)  **Splitting a Collection, Iterator or Array**  A common use case is to split a Collection, Iterator or Array from the message. In the sample below we simply use an [Expression](http://camel.apache.org/expression.html) to identify the value to split.   |  | | --- | | from("direct:splitUsingBody").split(body()).to("mock:result");    from("direct:splitUsingHeader").split(header("foo")).to("mock:result"); |   In Spring XML you can use the [Simple](http://camel.apache.org/simple.html) language to identify the value to split.   |  | | --- | | <split>     <simple>${body}</simple>     <to uri="mock:result"/>  </split>    <split>     <simple>${header.foo}</simple>     <to uri="mock:result"/>  </split> |   **Using Tokenizer from** [**Spring XML Extensions**](http://camel.apache.org/spring-xml-extensions.html)**\***  You can use the tokenizer expression in the Spring DSL to split bodies or headers using a token. This is a common use-case, so we provided a special **tokenizer** tag for this. In the sample below we split the body using a @ as separator. You can of course use comma or space or even a regex pattern, also set regex=true.   |  | | --- | | <camelContext xmlns="[http://camel.apache.org/schema/spring"](http://camel.apache.org/schema/spring%22)>      <route>          <from uri="direct:start"/>          <split>              <tokenize token="@"/>              <to uri="mock:result"/>          </split>      </route>  </camelContext> |   **What the Splitter returns**  **Camel 2.2 or older:** The [Splitter](http://camel.apache.org/splitter.html) will by default return the **last** splitted message.  **Camel 2.3 and newer** The [Splitter](http://camel.apache.org/splitter.html) will by default return the original input message.  **For all versions** You can override this by suppling your own strategy as an AggregationStrategy. There is a sample on this page (Split aggregate request/reply sample). Notice its the same strategy as the [Aggregator](http://camel.apache.org/aggregator.html) supports. This [Splitter](http://camel.apache.org/splitter.html) can be viewed as having a build in light weight [Aggregator](http://camel.apache.org/aggregator.html).  **Parallel execution of distinct 'parts'**  If you want to execute all parts in parallel you can use special notation of split() with two arguments, where the second one is a **boolean** flag if processing should be parallel. e.g.   |  | | --- | | XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");  from("activemq:my.queue").split(xPathBuilder, true).to("activemq:my.parts"); |   The boolean option has been refactored into a builder method parallelProcessing so its easier to understand what the route does when we use a method instead of true|false.   |  | | --- | | XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");  from("activemq:my.queue").split(xPathBuilder).parallelProcessing().to("activemq:my.parts"); |   **Stream based**  Splitting big XML payloads  The XPath engine in Java and [saxon](http://camel.apache.org/xquery.html) will load the entire XML content into memory. And thus they are not well suited for very big XML payloads. Instead you can use a custom [Expression](http://camel.apache.org/expression.html) which will iterate the XML payload in a streamed fashion. From Camel 2.9 onwards you can use the Tokenizer language which supports this when you supply the start and end tokens. From Camel 2.14, you can use the XMLTokenizer language which is specifically provided for tokenizing XML documents.  You can split streams by enabling the streaming mode using the streaming builder method.   |  | | --- | | from("direct:streaming").split(body().tokenize(",")).streaming().to("activemq:my.parts"); |   You can also supply your custom splitter to use with streaming like this:   |  | | --- | | import static org.apache.camel.builder.ExpressionBuilder.beanExpression;  from("direct:streaming")       .split(beanExpression(new MyCustomIteratorFactory(),  "iterator"))       .streaming().to("activemq:my.parts") |   **Streaming big XML payloads using Tokenizer language**  There are two tokenizers that can be used to tokenize an XML payload. The first tokenizer uses the same principle as in the text tokenizer to scan the XML payload and extract a sequence of tokens.  **Available as of Camel 2.9** If you have a big XML payload, from a file source, and want to split it in streaming mode, then you can use the Tokenizer language with start/end tokens to do this with low memory footprint.  StAX component  The Camel [StAX](http://camel.apache.org/stax.html) component can also be used to split big XML files in a streaming mode. See more details at [StAX](http://camel.apache.org/stax.html).  For example you may have a XML payload structured as follows   |  | | --- | | <orders>    <order>      <!-- order stuff here -->    </order>    <order>      <!-- order stuff here -->    </order>  ...    <order>      <!-- order stuff here -->    </order>  </orders> |   Now to split this big file using [XPath](http://camel.apache.org/xpath.html) would cause the entire content to be loaded into memory. So instead we can use the Tokenizer language to do this as follows:   |  | | --- | | from("file:inbox")    .split().tokenizeXML("order").streaming()       .to("activemq:queue:order"); |   In XML DSL the route would be as follows:   |  | | --- | | <route>    <from uri="file:inbox"/>    <split streaming="true">      <tokenize token="order" xml="true"/>      <to uri="activemq:queue:order"/>    </split>  </route> |   Notice the tokenizeXML method which will split the file using the tag name of the child node (more precisely speaking, the local name of the element without its namespace prefix if any), which mean it will grab the content between the <order> and </order> tags (incl. the tokens). So for example a splitted message would be as follows:   |  | | --- | | <order>    <!-- order stuff here -->  </order> |   If you want to inherit namespaces from a root/parent tag, then you can do this as well by providing the name of the root/parent tag:   |  | | --- | | <route>    <from uri="file:inbox"/>    <split streaming="true">      <tokenize token="order" inheritNamespaceTagName="orders" xml="true"/>      <to uri="activemq:queue:order"/>    </split>  </route> |   And in Java DSL its as follows:   |  | | --- | | from("file:inbox")    .split().tokenizeXML("order", "orders").streaming()       .to("activemq:queue:order"); |   Available as of Camel 2.13.1, you can set the above inheritNamsepaceTagName property to "\*" to include the preceding context in each token (i.e., generating each token enclosed in its ancestor elements). It is noted that each token must share the same ancestor elements in this case.  The above tokenizer works well on simple structures but has some inherent limitations in handling more complex XML structures.  **Available as of Camel 2.14**  The second tokenizer uses a StAX parser to overcome these limitations. This tokenizer recognizes XML namespaces and also handles simple and complex XML structures more naturally and efficiently.  To split using this tokenizer at {urn:shop}order, we can write   |  | | --- | | Namespaces ns = new Namespaces("ns1", "urn:shop");  ...  from("file:inbox")    .split().xtokenize("//ns1:order", 'i', ns).streaming()      .to("activemq:queue:order) |   Two arguments control the behavior of the tokenizer. The first argument specifies the element using a path notation. This path notation uses a subset of xpath with wildcard support. The second argument represents the extraction mode. The available extraction modes are:   |  |  | | --- | --- | | **mode** | **description** | | i | injecting the contextual namespace bindings into the extracted token (default) | | w | wrapping the extracted token in its ancestor context | | u | unwrapping the extracted token to its child content | | t | extracting the text content of the specified element |    Having an input XML   |  | | --- | | <m:orders xmlns:m="urn:shop" xmlns:cat="urn:shop:catalog">    <m:order><id>123</id><date>2014-02-25</date>...</m:order>  ... |   Each mode will result in the following tokens,   |  |  | | --- | --- | | i | <m:order xmlns:m="urn:shop" xmlns:cat="urn:shop:catalog"><id>123</id><date>2014-02-25</date>...</m:order> | | w | <m:orders xmlns:m="urn:shop" xmlns:cat="urn:shop:catalog">  <m:order><id>123</id><date>2014-02-25</date>...</m:order></m:orders> | | u | <id>123</id><date>2014-02-25</date>... | | t | 1232014-02-25... |    In XML DSL, the equivalent route would be written as follows:   |  | | --- | | <camelContext xmlns:ns1="urn:shop">    <route>      <from uri="file:inbox"/>      <split streaming="true">        <xtokenize>//ns1:order</xtokenize>        <to uri="activemq:queue:order"/>      </split>    </route>  </camelContext> |    or setting the extraction mode explicitly as   |  | | --- | | ...  <xtokenize mode="i">//ns1:order</xtokenize>  ... |   Note that this StAX based tokenizer's uses StAX Location API and requires a StAX Reader implementation (e.g., woodstox) that correctly returns the offset position pointing to the beginning of each event triggering segment (e.g., the offset position of '<' at each start and end element event). If you use a StAX Reader which does not implement that API correctly it results in invalid xml snippets after the split. For example the snippet could be wrong terminated:  <Start>...<</Start> .... <Start>...</</Start>  **Splitting files by grouping N lines together**  **Available as of Camel 2.10**  The [Tokenizer](http://camel.apache.org/tokenizer.html) language has a new option group that allows you to group N parts together, for example to split big files into chunks of 1000 lines.   |  | | --- | | from("file:inbox")    .split().tokenize("\n", 1000).streaming()       .to("activemq:queue:order"); |   And in XML DSL   |  | | --- | | <route>    <from uri="file:inbox"/>    <split streaming="true">      <tokenize token="\n" group="1000"/>      <to uri="activemq:queue:order"/>    </split>  </route> |   The group option is a number that must be a positive number that dictates how many groups to combine together. Each part will be combined using the token. So in the example above the message being sent to the activemq order queue, will contain 1000 lines, and each line separated by the token (which is a new line token). The output when using the group option is always a java.lang.String type.  **Specifying a custom aggregation strategy**  This is specified similar to the [Aggregator](http://camel.apache.org/aggregator.html).  **Specifying a custom ThreadPoolExecutor**  You can customize the underlying ThreadPoolExecutor used in the parallel splitter. In the Java DSL try something like this:   |  | | --- | | XPathBuilder xPathBuilder = new XPathBuilder("//foo/bar");    ExecutorService pool = ...    from("activemq:my.queue")      .split(xPathBuilder).parallelProcessing().executorService(pool)          .to("activemq:my.parts"); |   **Using a Pojo to do the splitting**  As the [Splitter](http://camel.apache.org/splitter.html) can use any [Expression](http://camel.apache.org/expression.html) to do the actual splitting we leverage this fact and use a **method** expression to invoke a [Bean](http://camel.apache.org/bean.html) to get the splitted parts. The [Bean](http://camel.apache.org/bean.html) should return a value that is iterable such as: java.util.Collection, java.util.Iterator or an array.  So the returned value, will then be used by Camel at runtime, to split the message.  Streaming mode and using pojo  When you have enabled the streaming mode, then you should return a Iterator to ensure streamish fashion. For example if the message is a big file, then by using an iterator, that returns a piece of the file in chunks, in the next method of the Iterator ensures low memory footprint. This avoids the need for reading the entire content into memory. For an example see the source code for the [TokenizePair](https://svn.apache.org/repos/asf/camel/trunk/camel-core/src/main/java/org/apache/camel/support/TokenPairExpressionIterator.java) implementation.  In the route we define the [Expression](http://camel.apache.org/expression.html) as a method call to invoke our [Bean](http://camel.apache.org/bean.html) that we have registered with the id mySplitterBean in the [Registry](http://camel.apache.org/registry.html).   |  | | --- | | from("direct:body")          // here we use a POJO bean mySplitterBean to do the split of the payload          .split().method("mySplitterBean", "splitBody")          .to("mock:result");  from("direct:message")          // here we use a POJO bean mySplitterBean to do the split of the message          // with a certain header value          .split().method("mySplitterBean", "splitMessage")          .to("mock:result"); |   And the logic for our [Bean](http://camel.apache.org/bean.html) is as simple as. Notice we use Camel [Bean Binding](http://camel.apache.org/bean-binding.html) to pass in the message body as a String object.   |  | | --- | | public class MySplitterBean {        /\*\*       \* The split body method returns something that is iteratable such as a java.util.List.       \*       \* @param body the payload of the incoming message       \* @return a list containing each part splitted       \*/      public List<String> splitBody(String body) {          // since this is based on an unit test you can of cause          // use different logic for splitting as Camel have out          // of the box support for splitting a String based on comma          // but this is for show and tell, since this is java code          // you have the full power how you like to split your messages          List<String> answer = new ArrayList<String>();          String[] parts = body.split(",");          for (String part : parts) {              answer.add(part);          }          return answer;      }        /\*\*       \* The split message method returns something that is iteratable such as a java.util.List.       \*       \* @param header the header of the incoming message with the name user       \* @param body the payload of the incoming message       \* @return a list containing each part splitted       \*/      public List<Message> splitMessage(@Header(value = "user") String header, @Body String body) {          // we can leverage the Parameter Binding Annotations          // <http://camel.apache.org/parameter-binding-annotations.html>          // to access the message header and body at same time,          // then create the message that we want, splitter will          // take care rest of them.          // \*NOTE\* this feature requires Camel version >= 1.6.1          List<Message> answer = new ArrayList<Message>();          String[] parts = header.split(",");          for (String part : parts) {              DefaultMessage message = new DefaultMessage();              message.setHeader("user", part);              message.setBody(body);              answer.add(message);          }          return answer;      }  } |   **Split aggregate request/reply sample**  This sample shows how you can split an [Exchange](http://camel.apache.org/exchange.html), process each splitted message, aggregate and return a combined response to the original caller using request/reply.  The route below illustrates this and how the split supports a **aggregationStrategy** to hold the in progress processed messages:   |  | | --- | | // this routes starts from the direct:start endpoint  // the body is then splitted based on @ separator  // the splitter in Camel supports InOut as well and for that we need  // to be able to aggregate what response we need to send back, so we provide our  // own strategy with the class MyOrderStrategy.  from("direct:start")      .split(body().tokenize("@"), new MyOrderStrategy())          // each splitted message is then send to this bean where we can process it          .to("bean:MyOrderService?method=handleOrder")          // this is important to end the splitter route as we do not want to do more routing          // on each splitted message      .end()      // after we have splitted and handled each message we want to send a single combined      // response back to the original caller, so we let this bean build it for us      // this bean will receive the result of the aggregate strategy: MyOrderStrategy      .to("bean:MyOrderService?method=buildCombinedResponse") |   And the OrderService bean is as follows:   |  | | --- | | public static class MyOrderService {        private static int counter;        /\*\*       \* We just handle the order by returning a id line for the order       \*/      public String handleOrder(String line) {          LOG.debug("HandleOrder: " + line);          return "(id=" + ++counter + ",item=" + line + ")";      }        /\*\*       \* We use the same bean for building the combined response to send       \* back to the original caller       \*/      public String buildCombinedResponse(String line) {          LOG.debug("BuildCombinedResponse: " + line);          return "Response[" + line + "]";      }  } |   And our custom **aggregationStrategy** that is responsible for holding the in progress aggregated message that after the splitter is ended will be sent to the **buildCombinedResponse** method for final processing before the combined response can be returned to the waiting caller.   |  | | --- | | /\*\*   \* This is our own order aggregation strategy where we can control   \* how each splitted message should be combined. As we do not want to   \* loos any message we copy from the new to the old to preserve the   \* order lines as long we process them   \*/  public static class MyOrderStrategy implements AggregationStrategy {        public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {          // put order together in old exchange by adding the order from new exchange            if (oldExchange == null) {              // the first time we aggregate we only have the new exchange,              // so we just return it              return newExchange;          }            String orders = oldExchange.getIn().getBody(String.class);          String newLine = newExchange.getIn().getBody(String.class);            LOG.debug("Aggregate old orders: " + orders);          LOG.debug("Aggregate new order: " + newLine);            // put orders together separating by semi colon          orders = orders + ";" + newLine;          // put combined order back on old to preserve it          oldExchange.getIn().setBody(orders);            // return old as this is the one that has all the orders gathered until now          return oldExchange;      }  } |   So lets run the sample and see how it works. We send an [Exchange](http://camel.apache.org/exchange.html) to the **direct:start** endpoint containing a IN body with the String value: A@B@C. The flow is:   |  | | --- | | HandleOrder: A  HandleOrder: B  Aggregate old orders: (id=1,item=A)  Aggregate new order: (id=2,item=B)  HandleOrder: C  Aggregate old orders: (id=1,item=A);(id=2,item=B)  Aggregate new order: (id=3,item=C)  BuildCombinedResponse: (id=1,item=A);(id=2,item=B);(id=3,item=C)  Response to caller: Response[(id=1,item=A);(id=2,item=B);(id=3,item=C)] |   **Stop processing in case of exception**  **Available as of Camel 2.1**  The [Splitter](http://camel.apache.org/splitter.html) will by default continue to process the entire [Exchange](http://camel.apache.org/exchange.html) even in case of one of the splitted message will thrown an exception during routing. For example if you have an [Exchange](http://camel.apache.org/exchange.html) with 1000 rows that you split and route each sub message. During processing of these sub messages an exception is thrown at the 17th. What Camel does by default is to process the remainder 983 messages. You have the chance to remedy or handle this in the AggregationStrategy.  But sometimes you just want Camel to stop and let the exception be propagated back, and let the Camel error handler handle it. You can do this in Camel 2.1 by specifying that it should stop in case of an exception occurred. This is done by the stopOnException option as shown below:   |  | | --- | | from("direct:start")      .split(body().tokenize(",")).stopOnException()          .process(new MyProcessor())          .to("mock:split"); |   And using XML DSL you specify it as follows:   |  | | --- | | <route>      <from uri="direct:start"/>      <split stopOnException="true">          <tokenize token=","/>          <process ref="myProcessor"/>          <to uri="mock:split"/>      </split>  </route> |   **Using onPrepare to execute custom logic when preparing messages**  **Available as of Camel 2.8**  See details at [Multicast](http://camel.apache.org/multicast.html)  **Sharing unit of work**  **Available as of Camel 2.8**  The [Splitter](http://camel.apache.org/splitter.html) will by default not share unit of work between the parent exchange and each splitted exchange. This means each sub exchange has its own individual unit of work.  For example you may have an use case, where you want to split a big message. And you want to regard that process as an atomic isolated operation that either is a success or failure. In case of a failure you want that big message to be moved into a [dead letter queue](http://camel.apache.org/dead-letter-channel.html). To support this use case, you would have to share the unit of work on the [Splitter](http://camel.apache.org/splitter.html).  Here is an example in Java DSL   |  | | --- | | errorHandler(deadLetterChannel("mock:dead").useOriginalMessage()          .maximumRedeliveries(3).redeliveryDelay(0));    from("direct:start")      .to("mock:a")      // share unit of work in the splitter, which tells Camel to propagate failures from      // processing the splitted messages back to the result of the splitter, which allows      // it to act as a combined unit of work      .split(body().tokenize(",")).shareUnitOfWork()          .to("mock:b")          .to("direct:line")      .end()      .to("mock:result");    from("direct:line")      .to("log:line")      .process(new MyProcessor())      .to("mock:line"); |   Now in this example what would happen is that in case there is a problem processing each sub message, the error handler will kick in (yes error handling still applies for the sub messages). **But** what doesn't happen is that if a sub message fails all redelivery attempts (its exhausted), then its **not** moved into that dead letter queue. The reason is that we have shared the unit of work, so the sub message will report the error on the shared unit of work. When the [Splitter](http://camel.apache.org/splitter.html) is done, it checks the state of the shared unit of work and checks if any errors occurred. And if an error occurred it will set the exception on the [Exchange](http://camel.apache.org/exchange.html) and mark it for rollback. The error handler will yet again kick in, as the [Exchange](http://camel.apache.org/exchange.html) has been marked as rollback and it had an exception as well. No redelivery attempts is performed (as it was marked for rollback) and the [Exchange](http://camel.apache.org/exchange.html) will be moved into the [dead letter queue](http://camel.apache.org/dead-letter-channel.html).  Using this from XML DSL is just as easy as you just have to set the shareUnitOfWork attribute to true:   |  | | --- | | <camelContext errorHandlerRef="dlc" xmlns="[http://camel.apache.org/schema/spring"](http://camel.apache.org/schema/spring%22)>      <!-- define error handler as DLC, with use original message enabled -->    <errorHandler id="dlc" type="DeadLetterChannel" deadLetterUri="mock:dead" useOriginalMessage="true">      <redeliveryPolicy maximumRedeliveries="3" redeliveryDelay="0"/>    </errorHandler>      <route>      <from uri="direct:start"/>      <to uri="mock:a"/>      <!-- share unit of work in the splitter, which tells Camel to propagate failures from           processing the splitted messages back to the result of the splitter, which allows           it to act as a combined unit of work -->      <split shareUnitOfWork="true">        <tokenize token=","/>        <to uri="mock:b"/>        <to uri="direct:line"/>      </split>      <to uri="mock:result"/>    </route>      <!-- route for processing each splitted line -->    <route>      <from uri="direct:line"/>      <to uri="log:line"/>      <process ref="myProcessor"/>      <to uri="mock:line"/>    </route>    </camelContext> |   Implementation of shared unit of work  So in reality the unit of work is not shared as a single object instance. Instead SubUnitOfWork is attached to their parent, and issues callback to the parent about their status (commit or rollback). This may be refactored in Camel 3.0 where larger API changes can be done.  **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*** 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 onwards and you should use this** [**Aggregator2**](http://camel.apache.org/aggregator2.html) **link instead.**  The [Aggregator](http://www.enterpriseintegrationpatterns.com/Aggregator.html) from the [EIP patterns](http://camel.apache.org/enterprise-integration-patterns.html) allows you to combine a number of messages together into a single message.  http://www.enterpriseintegrationpatterns.com/img/Aggregator.gif  A correlation [Expression](http://camel.apache.org/expression.html) 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](http://tmielke.blogspot.com/2009/01/using-camel-aggregator-correctly.html) 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**](http://camel.apache.org/fluent-builders.html)  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.IndexOutOfBoundsException: Index: 20, Size: 20  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](http://camel.apache.org/expression.html) languages such as [XPath](http://camel.apache.org/xpath.html), [XQuery](http://camel.apache.org/xquery.html), [SQL](http://camel.apache.org/sql.html) or various [Scripting Languages](http://camel.apache.org/scripting-languages.html).  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=&#39;APACHE&#39;])", String.class).batchSize(5).to("mock:result"); |   For further examples of this pattern in use you could look at the [junit test case](http://svn.apache.org/viewvc/camel/tags/camel-2.2.0/camel-core/src/test/java/org/apache/camel/processor/AggregatorTest.java?view=markup)  **Using the** [**Spring XML Extensions**](http://camel.apache.org/spring-xml-extensions.html)  The correlationExpression element is in Camel 2.0. For earliler 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](http://camel.apache.org/expression.html) for the correlation value used to aggregate messages together.  Error formatting macro: snippet: java.lang.IndexOutOfBoundsException: Index: 20, Size: 20  You can specify your own AggregationStrategy if you prefer as shown in the following example  Error formatting macro: snippet: java.lang.IndexOutOfBoundsException: Index: 20, Size: 20  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 incomming 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](http://camel.apache.org/batch-consumer.html). Then when enabled the [Aggregator](http://camel.apache.org/aggregator.html) will use the batch size determined by the [Batch Consumer](http://camel.apache.org/batch-consumer.html) in the message header CamelBatchSize. See more details at [Batch Consumer](http://camel.apache.org/batch-consumer.html). This can be used to aggregate all files consumed from a [File](http://camel.apache.org/file2.html) endpoint in that given poll. | | completionPredicate | null | Allows you to use a [Predicate](http://camel.apache.org/predicate.html) 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](http://camel.apache.org/expression.html) and an AggregationStrategy.   * The correlation [Expression](http://camel.apache.org/expression.html) 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  ExamplesDefault example By default Camel uses DefaultAggregationCollection and UseLatestAggregationStrategy, so this simple example will just keep the latest received exchange for the given correlation [Expression](http://camel.apache.org/expression.html):  Error formatting macro: snippet: java.lang.IndexOutOfBoundsException: Index: 20, Size: 20 Using PredicateAggregationCollection The PredicateAggregationCollection is an extension to DefaultAggregationCollection that uses a [Predicate](http://camel.apache.org/predicate.html) as well to determine the completion. For instance the [Predicate](http://camel.apache.org/predicate.html) 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.IndexOutOfBoundsException: Index: 20, Size: 20  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.IndexOutOfBoundsException: Index: 20, Size: 20  Then we setup the routing as follows:  Error formatting macro: snippet: java.lang.IndexOutOfBoundsException: Index: 20, Size: 20  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.IndexOutOfBoundsException: Index: 20, Size: 20 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.IndexOutOfBoundsException: Index: 20, Size: 20  Then we setup the routing as follows:  Error formatting macro: snippet: java.lang.IndexOutOfBoundsException: Index: 20, Size: 20  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.IndexOutOfBoundsException: Index: 20, Size: 20 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.IndexOutOfBoundsException: Index: 20, Size: 20  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.IndexOutOfBoundsException: Index: 20, Size: 20  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.IndexOutOfBoundsException: Index: 20, Size: 20 Using Batch Consumer **Available as of Camel 2.0**  The [Aggregator](http://camel.apache.org/aggregator.html) can work together with the [Batch Consumer](http://camel.apache.org/batch-consumer.html) to aggregate the total number of messages that the [Batch Consumer](http://camel.apache.org/batch-consumer.html) have reported. This allows you for instance to aggregate all files polled using the [File](http://camel.apache.org/file2.html) 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](http://camel.apache.org/batch-consumer.html) 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](http://camel.apache.org/getting-started.html), you may also find the [Architecture](http://camel.apache.org/architecture.html) useful particularly the description of [Endpoint](http://camel.apache.org/endpoint.html) and [URIs](http://camel.apache.org/uris.html). Then you could try out some of the [Examples](http://camel.apache.org/examples.html) first before trying this pattern out. | [**Overview**](http://camel.apache.org/overview.html)   * [Home](http://camel.apache.org/index.html) * [Download](http://camel.apache.org/download.html) * [Getting Started](http://camel.apache.org/getting-started.html) * [FAQ](http://camel.apache.org/faq.html)   [**Documentation**](http://camel.apache.org/documentation.html)   * [User Guide](http://camel.apache.org/user-guide.html) * [Manual](http://camel.apache.org/manual.html) * [Books](http://camel.apache.org/books.html) * [Tutorials](http://camel.apache.org/tutorials.html) * [Examples](http://camel.apache.org/examples.html) * [Cookbook](http://camel.apache.org/cookbook.html) * [Architecture](http://camel.apache.org/architecture.html) * [Enterprise Integration Patterns](http://camel.apache.org/enterprise-integration-patterns.html) * [DSL](http://camel.apache.org/dsl.html) * [Components](http://camel.apache.org/components.html) * [Data Format](http://camel.apache.org/data-format.html) * [Languages](http://camel.apache.org/languages.html) * [Security](http://camel.apache.org/security.html) * [Security Advisories](http://camel.apache.org/security-advisories.html)   **Search**  Top of Form    Bottom of Form |