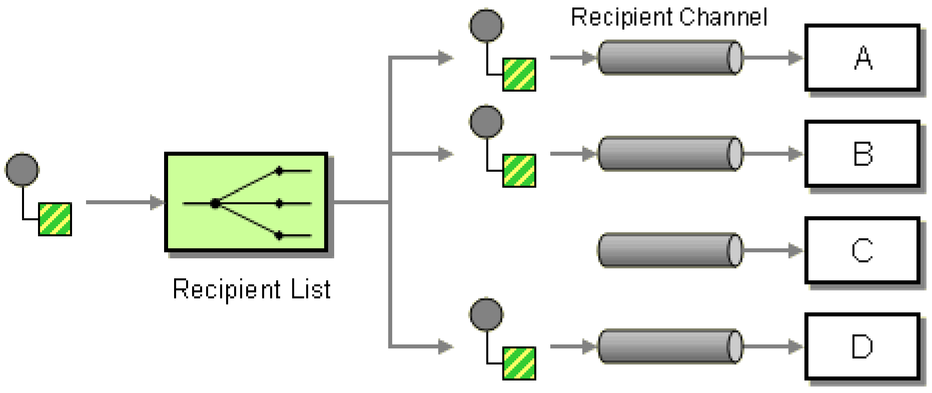
Recipient List

Recipient List

The Recipient List from the EIP patterns allows you to route messages to a number of dynamically specified recipients.



The recipients will receive a copy of the **same** Exchange, and Camel will execute them sequentially.

Options

confluenceTableSmall

Name	Default Value	Description
delimiter	,	Delimiter used if the Expression returned multiple endpoints (like "direct:foo,direct:bar"). From **Camel 2.13** onwards this can be disabled by setting delimiter to "false".
strategyRef		An AggregationStrategy that will assemble the replies from recipients into a single outgoing message from the Recipient List. By default Camel will use the last reply as the outgoing message. From **Camel 2.12** onwards you can also use a POJO as the `AggregationStrategy`, see the Aggregator 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 explicitly declare the method name to use, when using POJOs as the `AggregationStrategy`. See the Aggregator page for more details.
strategyMethodAllowNull	false	**Camel 2.12:** If this option is `false` then the aggregate method is not used if there was no data to enrich. If this option is `true` then `null` is used as the `oldExchange` (when no data to enrich), when using POJOs as the `AggregationStrategy`. See the Aggregator page for more details.
parallelProcessing	false	**Camel 2.2:** If enabled, messages are sent to the recipients concurrently. Note that the calling thread will still wait until all messages have been fully processed before it continues; it is the sending and processing of replies from recipients which happens in parallel.
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		**Camel 2.2:** A custom Thread Pool to use for parallel processing. Note that enabling this option implies parallel processing, so you need not enable that option as well.
stopOnException	false	**Camel 2.2:** Whether to immediately stop processing when an exception occurs. If disabled, Camel will send the message to all recipients regardless of any individual failures. You can process exceptions in an AggregationStrategy implementation, which supports full control of error handling.
ignoreInvalidEndpoints	false	**Camel 2.3:** Whether to ignore an endpoint URI that could not be resolved. If disabled, Camel will throw an exception identifying the invalid endpoint URI.
streaming	false	**Camel 2.5:** If enabled, Camel will process replies out-of-order - that is, in the order received in reply from each recipient. If disabled, Camel will process replies in the same order as specified by the Expression. So this specifies whether the response messages are aggregated as they come in, or in the exact order as the recipient list was evaluated. Only relevant if you enable parallelProcessing.
timeout		**Camel 2.5:** Specifies a processing timeout in milliseconds. If the Recipient List hasn't been able to send and process all replies within this timeframe, then the timeout triggers and the Recipient List breaks out, with message flow continuing to the next element. Note that if you provide a TimeoutAwareAggregationStrategy, its `timeout` method is invoked before breaking out. **Beware:** 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 caution. We may be able to improve this functionality in future Camel releases.

`onPrepareRef`		**Camel 2.8:** A custom Processor to prepare the copy of the Exchange each recipient will receive. This allows you to perform arbitrary transformations, such as deep-cloning the message payload (or any other custom logic).
`shareUnitOfWork`	`false`	**Camel 2.8:** Whether the unit of work should be shared. See the same option on Splitter for more details.
`cacheSize`	`1000`	**Camel 2.13.1/2.12.4:** Allows to configure the cache size for the `ProducerCache` which caches producers for reuse in the recipient list. Will by default use the default cache size which is 1000. Setting the value to -1 allows to turn off the cache completely.

Static Recipient List

The following example shows how to route a request from an input **queue:a** endpoint to a static list of destinations

Using Annotations
You can use the RecipientList Annotation on a POJO to create a Dynamic Recipient List. For more details see the Bean Integration.

Using the Fluent Builders**Using the Spring XML Extensions**

Dynamic Recipient List

Usually one of the main reasons for using the Recipient List pattern is that the list of recipients is dynamic and calculated at runtime. The following example demonstrates how to create a dynamic recipient list using an Expression (which in this case extracts a named header value dynamically) to calculate the list of endpoints which are either of type Endpoint or are converted to a String and then resolved using the endpoint URIs.

Using the Fluent Builders The above assumes that the header contains a list of endpoint URIs. The following takes a single string header and tokenizes it

Iteratable value

The dynamic list of recipients that are defined in the header must be iterable such as:

- `java.util.Collection`
- `java.util.Iterator`
- arrays
- `org.w3c.dom.NodeList`
- a single String with values separated by comma
- any other type will be regarded as a single value

Using the Spring XML Extensions For further examples of this pattern in action you could take a look at one of the junit test cases.

Using delimiter in Spring XML

In Spring DSL you can set the `delimiter` attribute for setting a delimiter to be used if the header value is a single String with multiple separated endpoints. By default Camel uses comma as delimiter, but this option lets you specify a custom delimiter to use instead.So if **myHeader** contains a String with the value `"activemq:queue:foo, activemq:topic:hello , log:bar"` then Camel will split the String using the delimiter given in the XML that was comma, resulting into 3 endpoints to send to. You can use spaces between the endpoints as Camel will trim the value when it lookup the endpoint to send to.

Note: In Java DSL you use the `tokenizer` to achieve the same. The route above in Java DSL:

In **Camel 2.1** its a bit easier as you can pass in the delimiter as 2nd parameter:

Sending to multiple recipients in parallel

Available as of Camel 2.2

The Recipient List now supports `parallelProcessing` that for example Splitter also supports. You can use it to use a thread pool to have concurrent tasks sending the Exchange to multiple recipients concurrently.

And in Spring XML it is an attribute on the recipient list tag.

Stop continuing in case one recipient failed

Available as of Camel 2.2

The Recipient List now supports `stopOnException` that for example Splitter also supports. You can use it to stop sending to any further recipients in case any recipient failed.

And in Spring XML its an attribute on the recipient list tag.

Note: You can combine `parallelProcessing` and `stopOnException` and have them both `true`.

Ignore invalid endpoints

Available as of Camel 2.3

The Recipient List now supports `ignoreInvalidEndpoints` (like the Routing Slip). You can use it to skip endpoints which are invalid.

And in Spring XML it is an attribute on the recipient list tag.

Then let us say the `myHeader` contains the following two endpoints `direct:foo,xxx:bar`. The first endpoint is valid and works. However the second one is invalid and will just be ignored. Camel logs at INFO level about it, so you can see why the endpoint was invalid.

Using custom `AggregationStrategy`

Available as of Camel 2.2

You can now use your own `AggregationStrategy` with the Recipient List. However this is rarely needed. What it is good for is that in case you are using Request Reply messaging then the replies from the recipients can be aggregated. By default Camel uses `UseLatestAggregationStrategy` which just keeps that last received reply. If you must remember all the bodies that all the recipients sent back, then you can use your own custom aggregator that keeps those. It is the same principle as with the Aggregator EIP so check it out for details.

And in Spring XML it is again an attribute on the recipient list tag.

Knowing which endpoint when using custom `AggregationStrategy`

Available as of Camel 2.12

When using a custom `AggregationStrategy` then the `aggregate` method is always invoked in sequential order (also if parallel processing is enabled) of the endpoints the Recipient List is using. However from Camel 2.12 onwards this is easier to know as the `newExchange` Exchange now has a property stored (key is `Exchange.RECIPIENT_LIST_ENDPOINT` with the uri of the Endpoint. So you know which endpoint you are aggregating from. The code block shows how to access this property in your Aggregator.

Using custom thread pool

Available as of Camel 2.2

A thread pool is only used for `parallelProcessing`. You supply your own custom thread pool via the `ExecutorServiceStrategy` (see Camel's Threading Model), the same way you would do it for the `aggregationStrategy`. By default Camel uses a thread pool with 10 threads (subject to change in future versions).

Using method call as recipient list

You can use a Bean to provide the recipients, for example:

And then `MessageRouter`:

When you use a Bean then do **not** use the `@RecipientList` annotation as this will in fact add yet another recipient list, so you end up having two. Do **not** do the following.

You should only use the snippet above (using `@RecipientList`) if you just route to a Bean which you then want to act as a recipient list.
So the original route can be changed to:

Which then would invoke the routeTo method and detect that it is annotated with `@RecipientList` and then act accordingly as if it was a recipient list EIP.

Using timeout

Available as of Camel 2.5

If you use `parallelProcessing` then you can configure a total `timeout` value in millis. Camel will then process the messages in parallel until the timeout is hit. This allows you to continue processing if one message consumer is slow. For example you can set a timeout value of 20 sec.

Tasks may keep running
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.

For example in the unit test below you can see that we multicast the message to 3 destinations. We have a timeout of 2 seconds, which means only the last two messages can be completed within the timeframe. This means we will only aggregate the last two which yields a result aggregation which outputs `"BC"`.

Timeout in other EIPs
This `timeout` feature is also supported by Splitter and both `multicast` and `recipientList`.

By default if a timeout occurs the `AggregationStrategy` is not invoked. However you can implement a special version

javaTimeoutAwareAggregationStrategyTimeoutAwareAggregationStrategy

This allows you to deal with the timeout in the `AggregationStrategy` if you really need to.

Timeout is total
The timeout is total, which means that after X time, Camel will aggregate the messages which have completed within the timeframe. The remainders will be cancelled. Camel will also only invoke the `timeout` method in the `TimeoutAwareAggregationStrategy` once, for the first index which caused the timeout.

Using onPrepare to execute custom logic when preparing messages

Available as of Camel 2.8

See details at Multicast

Using ExchangePattern in recipients

Available as of Camel 2.15

The recipient list will by default use the current Exchange Pattern. Though one can imagine use-cases where one wants to send a message to a recipient using a different exchange pattern. For example you may have a route that initiates as an InOnly route, but want to use InOut exchange pattern with a recipient list. To do this in earlier Camel releases, you would need to change the exchange pattern before the recipient list, or use onPrepare option to alter the pattern. From Camel 2.15 onwards, you can configure the exchange pattern directly in the recipient endpoints.

For example in the route below we pick up new files (which will be started as InOnly) and then route to a recipient list. As we want to use InOut with the ActiveMQ (JMS) endpoint we can now specify this using the exchangePattern=InOut option. Then the response from the JMS request/reply will then be continued routed, and thus the response is what will be stored in as a file in the outbox directory.

The recipient list will not alter the original exchange pattern. So in the example above the exchange pattern will still be InOnly when the message is routed to the file:outbox endpoint.

If you want to alter the exchange pattern permanently then use the .setExchangePattern option. See more details at Request Reply and Event Message.

Using This Pattern