# Load Balancer

## Load Balancer

The Load Balancer Pattern allows you to delegate to one of a number of endpoints using a variety of different load balancing policies.

## Built-in load balancing policies

Camel provides the following policies out-of-the-box:

| Policy | Description |
|---|---|
| Round Robin | The exchanges are selected from in a round robin fashion. This is a well known and classic policy, which spreads the load evenly. |
| Random | A random endpoint is selected for each exchange. |
| Sticky | Sticky load balancing using an Expression to calculate a correlation key to perform the sticky load balancing; rather like jsessionid in the web or JMSXGroupID in JMS. |
| Topic | Topic which sends to all destinations (rather like JMS Topics) |
| Failover | In case of failures the exchange will be tried on the next endpoint. |
| Weighted Round-Robin | **Camel 2.5:** The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to the others. In addition to the weight, endpoint selection is then further refined using **round-robin** distribution based on weight. |
| Weighted Random | **Camel 2.5:** The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to others.In addition to the weight, endpoint selection is then further refined using **random** distribution based on weight. |
| Custom | **Camel 2.8:** From Camel 2.8 onwards the preferred way of using a custom Load Balancer is to use this policy, instead of using the @deprecated `ref` attribute. |
| Circuit Breaker | **Camel 2.14:** Implements the Circuit Breaker pattern as described in "Release it!" book. |

Load balancing HTTP endpoints
If you are proxying and load balancing HTTP, then see this page for more details.

## Round Robin

The round robin load balancer is not meant to work with failover, for that you should use the dedicated **failover** load balancer. The round robin load balancer will only change to next endpoint per message.

The round robin load balancer is stateful as it keeps state of which endpoint to use next time.

**Using the Fluent Builders**{snippet:id=example|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/RoundRobinLoadBalanceTest.java}**Using the Spring configuration**
xml<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring"> <route> <from uri="direct:start"/> <loadBalance> <roundRobin/> <to uri="mock:x"/> <to uri="mock:y"/> <to uri="mock:z"/> </loadBalance> </route> </camelContext>

The above example loads balance requests from **direct:start** to one of the available **mock endpoint** instances, in this case using a round robin policy.
For further examples of this pattern look at this junit test case

## Failover

The `failover` load balancer is capable of trying the next processor in case an Exchange failed with an `exception` during processing.
You can constrain the `failover` to activate only when one exception of a list you specify occurs. If you do not specify a list any exception will cause fail over to occur. This balancer uses the same strategy for matching exceptions as the Exception Clause does for the **onException**.
Enable stream caching if using streams
If you use streaming then you should enable Stream caching when using the failover load balancer. This is needed so the stream can be re-read after failing over to the next processor.

Failover offers the following options:

| Option | Type | Default | Description |
|---|---|---|---|
| inheritErrorHandler | boolean | true | **Camel 2.3:** Whether or not the Error Handler configured on the route should be used. Disable this if you want failover to transfer immediately to the next endpoint. On the other hand, if you have this option enabled, then Camel will first let the Error Handler try to process the message. The Error Handler may have been configured to redeliver and use delays between attempts. If you have enabled a number of redeliveries then Camel will try to redeliver to the **same** endpoint, and only fail over to the next endpoint, when the Error Handler is exhausted. |
| maximumFailoverAttempts | int | -1 | **Camel 2.3:** A value to indicate after X failover attempts we should exhaust (give up). Use -1 to indicate never give up and continuously try to failover. Use 0 to never failover. And use e.g. 3 to failover at most 3 times before giving up. This option can be used whether or not roundRobin is enabled or not. |
| roundRobin | boolean | false | **Camel 2.3:** Whether or not the `failover` load balancer should operate in round robin mode or not. If not, then it will **always** start from the first endpoint when a new message is to be processed. In other words it restart from the top for every message. If round robin is enabled, then it keeps state and will continue with the next endpoint in a round robin fashion. When using round robin it will not *stick* to last known good endpoint, it will always pick the next endpoint to use. You can also enable sticky mode together with round robin, if so then it will pick the last known good endpoint to use when starting the load balancing (instead of using the next when starting). |
| sticky | boolean | false | **Camel 2.16:** Whether or not the failover load balancer should operate in sticky mode or not. If not, then it will always start from the first endpoint when a new message is to be processed. In other words it restart from the top for every message. If sticky is enabled, then it keeps state and will continue with the last known good endpoint. You can also enable sticky mode together with round robin, if so then it will pick the last known good endpoint to use when starting the load balancing (instead of using the next when starting). |

**Camel 2.2 or older behavior**
The current implementation of failover load balancer uses simple logic which **always** tries the first endpoint, and in case of an exception being thrown it tries the next in the list, and so forth. It has no state, and the next message will thus **always** start with the first endpoint.

**Camel 2.3 onwards behavior**
The `failover` load balancer now supports round robin mode, which allows you to failover in a round robin fashion. See the `roundRobin` option.
Redelivery must be enabled
In Camel 2.2 or older the failover load balancer requires you have enabled Camel Error Handler to use redelivery. In Camel 2.3 onwards this is not required as such, as you can mix and match. See the `inheritErrorHandler` option.

Here is a sample to failover only if a `IOException` related exception was thrown:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/FailOverNotCatchedExceptionTest.java}You can specify multiple exceptions to failover as the option is varargs, for instance:
java// enable redelivery so failover can react errorHandler(defaultErrorHandler().maximumRedeliveries(5)); from("direct:foo"). loadBalance().failover(IOException.class, MyOtherException.class) .to("direct:a", "direct:b");

## Using failover in Spring DSL

Failover can also be used from Spring DSL and you configure it as:
xml <route errorHandlerRef="myErrorHandler"> <from uri="direct:foo"/> <loadBalance> <failover> <exception>java.io.IOException</exception> <exception>com.mycompany.MyOtherException</exception> </failover> <to uri="direct:a"/> <to uri="direct:b"/> </loadBalance> </route>

## Using failover in round robin mode

An example using Java DSL:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/FailoverRoundRobinTest.java}And the same example using Spring XML:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/FailoverRoundRobinTest.xml}
Disabled inheritErrorHandler
You can configure `inheritErrorHandler=false` if you want to failover to the next endpoint as fast as possible. By disabling the Error Handler you ensure it does not *intervene* which allows the `failover` load balancer to handle failover asap. By also enabling `roundRobin` mode, then it will keep retrying until it success. You can then configure the `maximumFailoverAttempts` option to a high value to let it eventually exhaust (give up) and fail.

## Weighted Round-Robin and Random Load Balancing

**Available as of Camel 2.5**

In many enterprise environments where server nodes of unequal processing power & performance characteristics are utilized to host services and processing endpoints, it is frequently necessary to distribute processing load based on their individual server capabilities so that some endpoints are not unfairly burdened with requests. Obviously simple round-robin or random load balancing do not alleviate problems of this nature. A Weighted Round-Robin and/or Weighted Random load balancer can be used to address this problem.

The weighted load balancing policy allows you to specify a processing load distribution ratio for each server with respect to others. You can specify this as a positive processing weight for each server. A larger number indicates that the server can handle a larger load. The weight is

utilized to determine the payload distribution ratio to different processing endpoints with respect to others.
Disabled inheritErrorHandler
As of Camel 2.6, the Weighted Load balancer usage has been further simplified, there is no need to send in distributionRatio as a List<Integer>. It can be simply sent as a delimited String of integer weights separated by a delimiter of choice.

The parameters that can be used are

**In Camel 2.5**

| Option | Type | Default | Description |
|---|---|---|---|
| roundRobin | boolean | false | The default value for round-robin is false. In the absence of this setting or parameter the load balancing algorithm used is random. |
| distributionRatio | List<Integer> | none | The distributionRatio is a list consisting on integer weights passed in as a parameter. The distributionRatio must match the number of endpoints and/or processors specified in the load balancer list. In Camel 2.5 if endpoints do not match ratios, then a best effort distribution is attempted. |

**Available In Camel 2.6**

| Option | Type | Default | Description |
|---|---|---|---|
| roundRobin | boolean | false | The default value for round-robin is false. In the absence of this setting or parameter the load balancing algorithm used is random. |
| distributionRatio | String | none | The distributionRatio is a delimited String consisting on integer weights separated by delimiters for example "2,3,5". The distributionRatio must match the number of endpoints and/or processors specified in the load balancer list. |
| distributionRatioDelimiter | String | , | The distributionRatioDelimiter is the delimiter used to specify the distributionRatio. If this attribute is not specified a default delimiter "," is expected as the delimiter used for specifying the distributionRatio. |

## Using Weighted round-robin & random load balancing

**In Camel 2.5**

An example using Java DSL:
javaArrayList<integer> distributionRatio = new ArrayList<integer>(); distributionRatio.add(4); distributionRatio.add(2); distributionRatio.add(1); // round-robin from("direct:start") .loadBalance().weighted(true, distributionRatio) .to("mock:x", "mock:y", "mock:z"); //random from("direct:start") .loadBalance().weighted(false, distributionRatio) .to("mock:x", "mock:y", "mock:z");

And the same example using Spring XML:
xml <route> <from uri="direct:start"/> <loadBalance> <weighted roundRobin="false" distributionRatio="4 2 1"/> <to uri="mock:x"/> <to uri="mock:y"/> <to uri="mock:z"/> </loadBalance> </route>

**Available In Camel 2.6**

An example using Java DSL:
java// round-robin from("direct:start") .loadBalance().weighted(true, "4:2:1" distributionRatioDelimiter=":") .to("mock:x", "mock:y", "mock:z"); //random from("direct:start") .loadBalance().weighted(false, "4,2,1") .to("mock:x", "mock:y", "mock:z");

And the same example using Spring XML:
xml <route> <from uri="direct:start"/> <loadBalance> <weighted roundRobin="false" distributionRatio="4-2-1" distributionRatioDelimiter="-" /> <to uri="mock:x"/> <to uri="mock:y"/> <to uri="mock:z"/> </loadBalance> </route>

## Custom Load Balancer

You can use a custom load balancer (eg your own implementation) also.

An example using Java DSL:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/CustomLoadBalanceTest.java}And the same example using XML DSL:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/processor/SpringCustomRefLoadBalanceTest.xml}Notice in the XML DSL above we use <custom> which is only available in **Camel 2.8** onwards. In older releases you would have to do as follows instead:
xml <loadBalance ref="myBalancer"> <!-- these are the endpoints to balancer --> <to uri="mock:x"/> <to uri="mock:y"/> <to uri="mock:z"/> </loadBalance>

To implement a custom load balancer you can extend some support classes such as `LoadBalancerSupport` and `SimpleLoadBalancerSupport`. The former supports the asynchronous routing engine, and the latter does not. Here is an example:{snippet:id=e2|title=Custom load balancer implementation|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/CustomLoadBalanceTest.java}

## Circuit Breaker

The Circuit Breaker load balancer is a stateful pattern that monitors all calls for certain exceptions. Initially the Circuit Breaker is in closed state and passes all messages. If there are failures and the threshold is reached, it moves to open state and rejects all calls until halfOpenAfter timeout is reached. After this timeout is reached, if there is a new call, it will pass and if the result is success the Circuit Breaker will move to closed state, or to open state if there was an error.

When the circuit breaker is closed, it will throw a `java.util.concurrent.RejectedExecutionException`. This can then be caught to provide an alternate path for processing exchanges.

An example using Java DSL:
```
javafrom("direct:start") .onException(RejectedExecutionException.class) .handled(true) .to("mock:serviceUnavailable") .end()  .loadBalance()
.circuitBreaker(2, 1000L, MyCustomException.class) .to("mock:service") .end();
```

And the same example using Spring XML:
```
xml<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring"> <route> <from uri="direct:start"/> <onException>
<exception>java.util.concurrent.RejectedExecutionException</exception> <handled><constant>true</constant></handled> <to
uri="mock:serviceUnavailable"/> </onException>  <loadBalance> <circuitBreaker threshold="2" halfOpenAfter="1000">
<exception>MyCustomException</exception> </circuitBreaker> <to uri="mock:service"/> </loadBalance> </route> </camelContext>
```

Using This Pattern