Mock

Mock Component

Testing Summary Include

The Mock component provides a powerful declarative testing mechanism, which is similar to jMock in that it allows declarative expectations to be created on any Mock endpoint before a test begins. Then the test is run, which typically fires messages to one or more endpoints, and finally the expectations can be asserted in a test case to ensure the system worked as expected.

This allows you to test various things like:

- The correct number of messages are received on each endpoint,
- The correct payloads are received, in the right order,
- Messages arrive on an endpoint in order, using some Expression to create an order testing function,
- Messages arrive match some kind of Predicate such as that specific headers have certain values, or that parts of the messages match some predicate, such as by evaluating an XPath or XQuery Expression.

Note that there is also the Test endpoint which is a Mock endpoint, but which uses a second endpoint to provide the list of expected message bodies and automatically sets up the Mock endpoint assertions. In other words, it's a Mock endpoint that automatically sets up its assertions from some sample messages in a File or database, for example.

Mock endpoints keep received Exchanges in memory indefinitely
Remember that Mock is designed for testing. When you add Mock endpoints to a route, each Exchange sent to the endpoint will be stored (to allow for later validation) in memory until explicitly reset or the JVM is restarted. If you are sending high volume and/or large messages, this may cause excessive memory use. If your goal is to test deployable routes inline, consider using NotifyBuilder or AdviceWith in your tests instead of adding Mock endpoints to routes directly.

From Camel 2.10 onwards there are two new options `retainFirst`, and `retainLast` that can be used to limit the number of messages the Mock endpoints keep in memory.

URI format

mock:someName[?options]

Where **someName** can be any string that uniquely identifies the endpoint.

You can append query options to the URI in the following format, `?option=value&option=value&...`

Options

confluenceTableSmall

Option	Default	Description
`reportGroup`	`null`	A size to use a throughput logger for reporting
`retainFirst`		**Camel 2.10:** To only keep first X number of messages in memory.
`retainLast`		**Camel 2.10:** To only keep last X number of messages in memory.

Simple Example

Here's a simple example of Mock endpoint in use. First, the endpoint is resolved on the context. Then we set an expectation, and then, after the test has run, we assert that our expectations have been met.

MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class); resultEndpoint.expectedMessageCount(2); // send some messages ... // now lets assert that the mock:foo endpoint received 2 messages resultEndpoint.assertIsSatisfied();

You typically always call the assertIsSatisfied() method to test that the expectations were met after running a test.

Camel will by default wait 10 seconds when the `assertIsSatisfied()` is invoked. This can be configured by setting the `setResultWaitTime(millis)` method.

Using assertPeriod

Available as of Camel 2.7
When the assertion is satisfied then Camel will stop waiting and continue from the `assertIsSatisfied` method. That means if a new message arrives on the mock endpoint, just a bit later, that arrival will not affect the outcome of the assertion. Suppose you do want to test that no new messages arrives after a period thereafter, then you can do that by setting the `setAssertPeriod` method, for example:

MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class); resultEndpoint.setAssertPeriod(5000); resultEndpoint.expectedMessageCount(2); // send some messages ... // now lets assert that the mock:foo endpoint received 2 messages resultEndpoint.assertIsSatisfied();

Setting expectations

You can see from the javadoc of MockEndpoint the various helper methods you can use to set expectations. The main methods are as follows:

confluenceTableSmall

Method	Description
expectedMessageCount(int)	To define the expected message count on the endpoint.
expectedMinimumMessageCount(int)	To define the minimum number of expected messages on the endpoint.
expectedBodiesReceived(...)	To define the expected bodies that should be received (in order).
expectedHeaderReceived(...)	To define the expected header that should be received
expectsAscending(Expression)	To add an expectation that messages are received in order, using the given Expression to compare messages.
expectsDescending(Expression)	To add an expectation that messages are received in order, using the given Expression to compare messages.
expectsNoDuplicates(Expression)	To add an expectation that no duplicate messages are received; using an Expression to calculate a unique identifier for each message. This could be something like the `JMSMessageID` if using JMS, or some unique reference number within the message.

Here's another example:

resultEndpoint.expectedBodiesReceived("firstMessageBody", "secondMessageBody", "thirdMessageBody");

Adding expectations to specific messages

In addition, you can use the message(int messageIndex) method to add assertions about a specific message that is received.

For example, to add expectations of the headers or body of the first message (using zero-based indexing like `java.util.List`), you can use the following code:

resultEndpoint.message(0).header("foo").isEqualTo("bar");

There are some examples of the Mock endpoint in use in the camel-core processor tests.

Mocking existing endpoints

Available as of Camel 2.7

Camel now allows you to automatically mock existing endpoints in your Camel routes.

How it works
Important: The endpoints are still in action. What happens differently is that a Mock endpoint is injected and receives the message first and then delegates the message to the target endpoint. You can view this as a kind of intercept and delegate or endpoint listener.

Suppose you have the given route below:

{snippet:id=route|title=Route|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/interceptor/AdviceWithMockEndpointsTest.java}

You can then use the `adviceWith` feature in Camel to mock all the endpoints in a given route from your unit test, as shown below:

{snippet:id=e1|title=adviceWith mocking all endpoints|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/interceptor/AdviceWithMockEndpointsTest.java}

Notice that the mock endpoints is given the uri `mock:<endpoint>`, for example `mock:direct:foo`. Camel logs at `INFO` level the endpoints being mocked:

INFO Adviced endpoint [direct://foo] with mock endpoint [mock:direct:foo] Mocked endpoints are without parameters
Endpoints which are mocked will have their parameters stripped off. For example the endpoint "log:foo?showAll=true" will be mocked to the following endpoint "mock:log:foo". Notice the parameters have been removed.

Its also possible to only mock certain endpoints using a pattern. For example to mock all `log` endpoints you do as shown:

{snippet:id=e2|lang=java|title=adviceWith mocking only log endpoints using a pattern|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/interceptor/AdviceWithMockEndpointsTest.java}

The pattern supported can be a wildcard or a regular expression. See more details about this at Intercept as its the same matching function used by Camel.

Mind that mocking endpoints causes the messages to be copied when they arrive on the mock.
That means Camel will use more memory. This may not be suitable when you send in a lot of messages.

Mocking existing endpoints using the `camel-test` component

Instead of using the `adviceWith` to instruct Camel to mock endpoints, you can easily enable this behavior when using the `camel-test` Test Kit.
The same route can be tested as follows. Notice that we return `"*"` from the `isMockEndpoints` method, which tells Camel to mock all endpoints.
If you only want to mock all `log` endpoints you can return `"log*"` instead.

{snippet:id=e1|lang=java|title=isMockEndpoints using camel-test kit|url=camel/trunk/components/camel-test/src/test/java/org/apache/camel/test/patterns
/IsMockEndpointsJUnit4Test.java}

Mocking existing endpoints with XML DSL

If you do not use the `camel-test` component for unit testing (as shown above) you can use a different approach when using XML files for routes.
The solution is to create a new XML file used by the unit test and then include the intended XML file which has the route you want to test.

Suppose we have the route in the `camel-route.xml` file:

{snippet:id=e1|lang=xml|title=camel-route.xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/mock/camel-route.xml}

Then we create a new XML file as follows, where we include the `camel-route.xml` file and define a spring bean with the class `org.apache.camel.impl.InterceptSendToMockEndpointStrategy` which tells Camel to mock all endpoints:

{snippet:id=e1|lang=xml|title=test-camel-route.xml|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/mock
/InterceptSendToMockEndpointStrategyTest.xml}

Then in your unit test you load the new XML file (`test-camel-route.xml`) instead of `camel-route.xml`.

To only mock all Log endpoints you can define the pattern in the constructor for the bean:

xml<bean id="mockAllEndpoints" class="org.apache.camel.impl.InterceptSendToMockEndpointStrategy"> <constructor-arg index="0" value="log*"/> <
/bean>

Mocking endpoints and skip sending to original endpoint

Available as of Camel 2.10

Sometimes you want to easily mock and skip sending to a certain endpoints. So the message is detoured and send to the mock endpoint only. From Camel 2.10 onwards you can now use the `mockEndpointsAndSkip` method using AdviceWith or the Test Kit. The example below will skip sending to the two endpoints `"direct:foo"`, and `"direct:bar"`.

{snippet:id=e1|lang=java|title=adviceWith mock and skip sending to endpoints|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor
/interceptor/AdviceWithMockMultipleEndpointsWithSkipTest.java}

The same example using the Test Kit

{snippet:id=e1|lang=java|title=isMockEndpointsAndSkip using camel-test kit|url=camel/trunk/components/camel-test/src/test/java/org/apache/camel/test
/patterns/IsMockEndpointsAndSkipJUnit4Test.java}

Limiting the number of messages to keep

Available as of Camel 2.10

The Mock endpoints will by default keep a copy of every Exchange that it received. So if you test with a lot of messages, then it will consume memory.
From Camel 2.10 onwards we have introduced two options `retainFirst` and `retainLast` that can be used to specify to only keep N'th of the first and
/or last Exchanges.

For example in the code below, we only want to retain a copy of the first 5 and last 5 Exchanges the mock receives.

MockEndpoint mock = getMockEndpoint("mock:data"); mock.setRetainFirst(5); mock.setRetainLast(5); mock.expectedMessageCount(2000); ... mock.
assertIsSatisfied();

Using this has some limitations. The `getExchanges()` and `getReceivedExchanges()` methods on the `MockEndpoint` will return only the retained copies of the Exchanges. So in the example above, the list will contain 10 Exchanges; the first five, and the last five.
The `retainFirst` and `retainLast` options also have limitations on which expectation methods you can use. For example the expectedXXX methods that work on message bodies, headers, etc. will only operate on the retained messages. In the example above they can test only the expectations on the 10 retained messages.

Testing with arrival times

Available as of Camel 2.7

The Mock endpoint stores the arrival time of the message as a property on the Exchange.

Date time = exchange.getProperty(Exchange.RECEIVED_TIMESTAMP, Date.class);

You can use this information to know when the message arrived on the mock. But it also provides foundation to know the time interval between the previous and next message arrived on the mock. You can use this to set expectations using the `arrives` DSL on the Mock endpoint.

For example to say that the first message should arrive between 0-2 seconds before the next you can do:

mock.message(0).arrives().noLaterThan(2).seconds().beforeNext();

You can also define this as that 2nd message (0 index based) should arrive no later than 0-2 seconds after the previous:

mock.message(1).arrives().noLaterThan(2).seconds().afterPrevious();

You can also use between to set a lower bound. For example suppose that it should be between 1-4 seconds:

mock.message(1).arrives().between(1, 4).seconds().afterPrevious();

You can also set the expectation on all messages, for example to say that the gap between them should be at most 1 second:

mock.allMessages().arrives().noLaterThan(1).seconds().beforeNext(); time units
In the example above we use `seconds` as the time unit, but Camel offers `milliseconds`, and `minutes` as well.

Endpoint See Also

- Spring Testing
- Testing