NotifyBuilder

NotifyBuilder

Available from Camel 2.2

The **`NotifyBuilder`** is a builder from the **`org.apache.camel.builder`** package which allows you to build expressions and then test or wait for that condition to occur. The expressions is based around notifications about Exchange being routed. So what does that mean? It means that you can build an expressions which can tell you when Camel is finished with routing 5 messages etc.

You may want to use this when testing a route which you cannot or will not use Mocks.

Suppose we have a very simple route:

```
from("jms:queue:quotes")
    .to("bean:quotes");
```

Now you want to test this route without using mocks or the likes. Imagine the route being more complex and a production ready route.
We want to test that it could process a message send to that queue. By using the NotifyBuilder we can build an expression which *expresses when that condition occurred*.

```
NotifyBuilder notify = new NotifyBuilder().whenDone(1).create();

// now use some API to send a message etc. Maybe you cannot use Camel's ProducerTemplate
// now we want to wait until the message has been routed and completed

boolean done = notify.matches(10, TimeUnit.SECONDS);
assertTrue("Should be done", done);

// now maybe use some API to see that the message did as expected
```

This is a very basic example with a simple builder expression. What we said that we want it to match when any Exchange is done. The builder have many more methods to set more complex expressions, which even can be stacked using **`and`**, **`or`**, **`not`** operations.

Methods

These methods is for building the expression:

Method	Description
`from(endpointUri)`	Matches only when Exchanges are incoming from that particular endpoint. The **`endpointUri`** can be a pattern, which is the same pattern matching used by Intercept.
`fromRoute(routeId)`	**Camel 2.4:** Matches only when Exchanges are incoming from that particular route. The **`routeId`** can be a pattern, which is the same pattern matching used by Intercept.
`filter(predicate)`	Filters out unwanted Exchanges (only messages passing (true) the predicate is used).
`wereSentTo (endpointUri)`	**Camel 2.9:** Matches only when Exchanges has at any point been sent to the given endpoint. The **`endpointUri`** can be a pattern, which is the same pattern matching used by Intercept.
`whenReceived(number)`	Matches when X number or more messages has been received.
`whenDone(number)`	Matches when X number or more messages is done.
`whenDoneByIndex (index)`	**Camel 2.8:** Matches when the N'th (index) message is done.
`whenComplete(number)`	Matches when X number or more messages is complete.
`whenFailed(number)`	Matches when X number or more messages is failed.
`whenExactlyDone (number)`	Matches when exactly X number of messages is done.
`whenExactlyComplete (number)`	Matches when exactly X number of messages is complete.
`whenExactlyFailed (number)`	Matches when exactly X number of messages is failed.

`whenBodiesReceived(bodies)`	Matches when the message bodies has been received in the same order. This method is non strict which means that it will disregard any additional received messages.
`whenExactBodiesReceived(bodies)`	Matches when the message bodies has been received in the same order. This method is strict which means the exact number of message bodies is expected.
`whenBodiesDone(bodies)`	Matches when the message bodies are done in the same order. This method is non strict which means that it will disregard any additional done messages.
`whenExactBodiesDone(bodies)`	Matches when the message bodies are done in the same order. This method is strict which means the exact number of message bodies is expected.
`whenAnyReceivedMatches(predicate)`	Matches if any one of the received messages matched the Predicate.
`whenAllReceivedMatches(predicate)`	Matches only when all of the received messages matched the Predicate.
`whenAnyDoneMatches(predicate)`	Matches if any one of the done messages matched the Predicate.
`whenAllDoneMatches(predicate)`	Matches only when all of the done messages matched the Predicate.
`whenReceivedSatisfied(mock)`	Matches if the Mock is satisfied for received messages. Is used for fine grained matching by setting the expectations on the Mock which already have a great library for doing so.
`whenReceivedNotSatisfied(mock)`	Matches if the Mock is **not** satisfied for received messages. Is used for fine grained matching by setting the expectations on the Mock which already have a great library for doing so.
`whenDoneSatisfied(mock)`	Matches if the Mock is satisfied for messages done. Is used for fine grained matching by setting the expectations on the Mock which already have a great library for doing so.
`whenDoneNotSatisfied(mock)`	Matches if the Mock is **not** satisfied for messages done. Is used for fine grained matching by setting the expectations on the Mock which already have a great library for doing so.
`and`	Appends an additional expressions using the **`and`** operator.
`or`	Appends an additional expressions using the **`or`** operator.
`not`	Appends an additional expressions using the **`not`** operator.

And these methods is for using the builder after creating the expression:

Method	Description
`create()`	Creates the builder expression. After you have ***created*** it you can use the `matches` methods.
`matches()`	Does the builder match currently. This operation returns immediately. This method is to be used ***after*** you have created the expression.
`matches(timeout, TimeUnit)`	Wait until the builder matches or timeout. This method is to be used ***after*** you have created the expression.
`matchesMockWaitTime`	**Camel 2.6:** Wait until the builder matches or timeout. The timeout value used is based on the ***highest result wait time*** configured on any of mock endpoints being used. If no mock endpoint was used, then the default timeout value is **`10`** seconds. This method is convenient to use in unit tests when you use mocks. Then you don't have to specify the timeout value explicit.
`reset()`	**Camel 2.3:** Resets the notifier.

We will most likely add additional methods in the future, so check out the **`NotifyBuilder`** for latest and greatest methods.

Difference Between `Done` and `Completed`

The difference between **done** and **`completed`** is that done can also include failed messages, where as completed is only successful processed messages.

Examples

```
NotifyBuilder notify = new NotifyBuilder(context)
    .from("direct:foo").whenDone(5)
    .create();
```

Here we want to match when the **`direct:foo`** endpoint have done 5 messages.

You may also want to be notified when an message is done by the index, for example the very first message. To do that you can simply do:

```
NotifyBuilder notify = new NotifyBuilder(context)
    .whenDoneByIndex(0)
    .create();
```

This ensures that the notifier only matches exactly when the first message is done.

If you use **whenDone(1)** instead, then the notifier matches when at least one message is done. There could be use cases where **whenDone(1)** would match even if the first message hasn't been done yet, as other message in between could be done ahead of the first message. That is why **whenDoneByIndex** was introduced in **Camel 2.8** to support this scenario.

```
NotifyBuilder notify = new NotifyBuilder(context)
    .from("direct:foo").filter(body().contains("test")).whenDone(5)
    .create();
```

Here we want to match when the **direct:foo** endpoint have done 5 messages which contains the word '**test**' in the body.
The filter accepts a Predicate so you can use XPath, Bean, Simple and whatnot.

```
NotifyBuilder notify = new NotifyBuilder(context)
    .from("jms:*").whenDone(1)
    .create();
```

Here we just say that at least one message should be done received from any JMS endpoint (notice the wildcard matching).

```
NotifyBuilder notify = new NotifyBuilder(context)
    .fromRoute("myCoolRoutes*").whenDone(3)
    .create();
```

Here, we just say that at least three message should be done received from any of **myCoolRoutes** (notice the wildcard matching).

```
NotifyBuilder notify = new NotifyBuilder(context)
    .from("direct:foo").whenDone(5)
    .and().from("direct:bar").whenDone(7)
    .create();
```

Here both **5** foo messages and **7** bar messages must be done. Notice the use of the **and** operator.

```
NotifyBuilder notify = new NotifyBuilder(context)
    .from("direct:foo").whenBodiesReceived("Hello World", "Bye World")
    .create();
```

Here we expect to receive two messages with **Hello World** and **Bye World**.

```
NotifyBuilder notify = new NotifyBuilder(context)
    .whenAnyReceivedMatches(body().contains("Camel"))
    .create();
```

Here we want to match when we have received a message which contains Camel in the body.

```
// lets use a mock to set the expressions as it got many great assertions for that
// notice we use mock:assert which does NOT exist in the route, its just a pseudo name
MockEndpoint mock = getMockEndpoint("mock:assert");
mock.expectedBodiesReceivedInAnyOrder("Hello World", "Bye World", "Hi World");

NotifyBuilder notify = new NotifyBuilder(context)
    .from("direct:foo").whenReceivedSatisfied(mock)
    .create();
```

Now it bring powers to the table. We combine a mock with the builder. We use the mock to set fine grained expectations such as we should receive 3 messages in any order. Then using the builder we can tell that those messages should be received from the **`direct:foo`** endpoint. You can combine multiple expressions as much as you like. However we suggest to use the mock for fine grained expectations that you may already know how to use. You can also specify that the Exchanges must have been sent to a given endpoint.

For example in the following we expect the message to be sent to **`mock:bar`**

```
NotifyBuilder notify = new NotifyBuilder(context)
    .wereSentTo("mock:bar")
    .create();
```

You can combine this with any of the other expectations, such as, to only match if 3+ messages are done, and were sent to the **`mock:bar`** endpoint:

```
NotifyBuilder notify = new NotifyBuilder(context)
    .whenDone(3).wereSentTo("mock:bar")
    .create();
```

You can add additional **`wereSentTo`**'s, such as the following two:

```
NotifyBuilder notify = new NotifyBuilder(context)
    .wereSentTo("activemq:queue:foo").wereSentTo("activemq:queue:bar")
    .create();
```

As well as you can expect a number of messages to be done, and a message to fail, which has to be sent to another endpoint:

```
NotifyBuilder notify = new NotifyBuilder(context)
    .whenDone(3).wereSentTo("activemq:queue:goodOrder")
    .and().whenFailed(1).wereSentTo("activemq:queue:badOrder")
    .create();
```