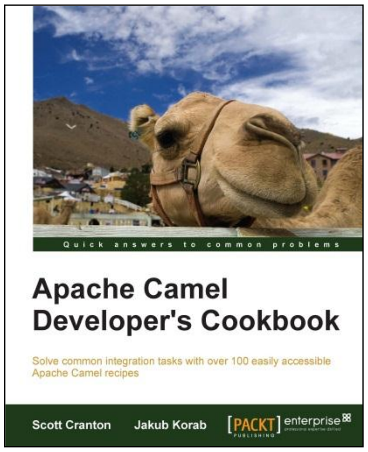
Apache Camel Developer's Cookbook

Scott Cranton

Jakub Korab



Chapter No. 9
"Testing"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.9 "Testing"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Scott Cranton is an open source software contributor and evangelist. He has been working with Apache Camel since the release of version 1.5 almost 5 years ago, and has over 20 years of commercial experience in middleware software as a developer, architect, and consultant. During his time at FuseSource, and now Red Hat, he has worked closely with many core committers for Apache Camel, ActiveMQ, ServiceMix, Karaf, and CXF. He has also helped many companies successfully create and deploy large and complex integration and messaging systems using Camel and other open source projects.

He divides his professional time between hacking code, delivering webinars on using Camel and open source, and helping companies to learn how to use Camel to solve their integration problems.

I want to thank my amazing wife, Mary Elizabeth, for putting up with me these many years, and always answering the phone when I'd call late at night from the office while I've got a compile going. This book would not have been possible without her always being there for me no matter what. To my three wonderful children, Gilbert, Eliza, and Lucy, who always make me smile especially during crazy weekend writing sessions when they'd want me to take a break, "… but Dad, it's the weekend…" I love you all!

Jakub Korab is a consulting software engineer specializing in integration and messaging. With a formal background in software engineering and distributed systems, in the 14 years that he has worked in software across the telecoms, financial services, and banking industries, he naturally gravitated from web development towards systems integration. When he discovered Apache Camel, it became apparent to him how much time and effort it could have saved him in the past compared to writing bespoke integration code, and he has not looked back since.

Over the years, working as a consultant, he has helped dozens of clients build scalable, fault-tolerant, and performant systems integrations. He currently runs his own specialist consultancy, Ameliant, which focuses on systems integration and messaging using a stack of integration products from the Apache Software Foundation, of which Camel is a corner stone.

When not gluing systems together, you will find him spending time with his young family, and far too infrequently kitesurfing or skiing—neither of which he gets much chance to do in his adopted home, London.

The writing of this book has taken place against the background of starting a new company, a huge amount of work travel, a quickly growing baby, house move, and hundreds of little distractions that get in the way of sitting down in what is left of the day to put pen to paper. It could never have happened without the love, support, and understanding of my wife, Anne-Marie. It has been a team effort. Thank you.

Also to my little girl, Alex, for helping me keep it all in perspective.

Apache Camel Developer's Cookbook

Apache Camel is a Java framework for building system integrations.

Why, you may well ask, does anyone need such a framework? System integration is pretty much a solved problem. After all, we have been connecting various frontends to web services, message brokers, and databases for years! Surely this is a well-understood domain that requires no further abstractions.

Not quite.

Apache Camel, since its release in 2007, has disrupted the integration market much like the Spring Framework disrupted the Java EE market back in 2003. Camel enables a new way of doing, and thinking about, system integrations that results in much cleaner, easier to understand code, which in turn results in less work, less bugs, and easier maintenance. These are big claims, and to validate them you only need to look at the large and active Apache Camel community, the growing number of commercial integration products based on Camel, and the talks on Camel that appear at most middleware developer conferences to feel that there is a good buzz around Camel, and for very good reason.

This book is targeted at readers who already have some familiarity with Camel, and are looking for tips on how Camel may be able to better help them solve more complex integration challenges. This book is structured as a series of over 100 how-to recipes, including step-by-step instructions on using Camel to solve common integration tasks. Each recipe includes a brief explanation of what Camel is doing internally, and references on where to find more information for those who want to dig deeper.

This book may not be a good introduction/beginner book about Camel, though if you have familiarity with other integration technologies, and learn well by doing, you may find this book's recipe approach helpful. This book does not spend a lot of time explaining Camel concepts in great depth.

For readers looking for more conceptual coverage of Camel (with lots of code examples), we would recommend reading the excellent book *Camel in Action* by Claus Ibsen and Jonathan Anstey, published by Manning. For a more introductory guide, look at *Instant Apache Camel Message Routing* by Bilgin Ibryam, published by Packt Publishing. The Apache Camel website (`http://camel.apache.org`) is the authoritative site on Camel, with a long list of articles and documentation that will help you on your journey of using Camel.

What is Camel?

This section provides a quick overview of what Camel is, and why it was created. Its goal is to help remind the reader of the core concepts used within Camel, and to help the reader understand how the authors define those concepts. It is not intended as a comprehensive introduction to Camel. Hopefully, it will act as a quick reference for Camel concepts as you use the various recipes contained within this book.

Integrating systems is hard work. It is hard because the developers doing the integration work must understand and how the endpoint systems expose themselves to external systems, how to transform and route the data records (messages) from each of the systems. They must also have a working knowledge of the ever growing number of technologies used in transporting, routing, and manipulating those messages. What makes it more challenging is that the systems you are integrating with were probably written by different teams of developers, at different times, and are probably still changing even while you are trying to integrate them. This is equivalent to connecting two cars while they are driving down the highway.

Traditional system integrations, in the way that we have built them in the past decades, require a lot of code to be created that has *absolutely nothing* to do with the higher-level integration problem trying to be solved. The vast majority of this is boilerplate code dealing with common, repetitive tasks of setting up and tearing down libraries for the messaging transports and processing technologies such as filesystems, SOAP, JMS, JDBC, socket-level I/O, XSLT, templating libraries, among others. These mechanical concerns are repeated over and over again in every single integration project's code base.

In early 2000, there were many people researching and cataloging software patterns within many of these projects, and this resulted in the excellent book *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* by Gregor Hohpe and Bobby Woolf, published by Addison Wesley. This catalog of integration patterns, EIPs for short, can be viewed at `http://www.enterpriseintegrationpatterns.com`. These patterns include classics such as the Content Based Router, Splitter, and Filter. The EIP book also introduces a model of how data moves from one system to another that is independent of the technology doing the work. These named concepts have become a common language for all integration architects and developers making it easier to express what an integration should do without getting lost in how to implement that integration.

Camel embraces these EIP concepts as core constructs within its framework, providing an executable version of those concepts that are independent of the mechanics of the underlying technology actually doing the work. Camel adds in abstractions such as **Endpoint URIs** (Uniform Resource Identifier) that work with **Components** (Endpoint factories), which allows developers to specify the desired technology to be used in connecting to an endpoint system without getting lost in the boilerplate code required to use that technology. Camel provides an integration, **domain-specific language (DSL)** for defining integration logic that is adapted to many programming languages (Java, Groovy, Scala, and so on) and frameworks (Spring, OSGi Blueprint, and so on), so that the developer can write code that is an English-like expression of the integration using EIP concepts. For example:

```
consume from some endpoint,
split the messages
  based on an expression, and
  send those split messages to some other endpoint
```

Let us look at a concrete example to show you how to use Camel.

Imagine that your boss comes to you, asking you to solve an integration problem for your project. You are asked to: poll a specific directory for new XML fi les every minute, split those XML files that contain many repeating elements into individual records/messages (think line items), and send each of those individual records to a JMS queue for processing by another system. Oh, and make sure that the code can retry if it hits any issues. Also, it is likely the systems will change shortly, so make sure the code is fl exible enough to handle changes, but we do not know what those changes might look like. Sound familiar?

Before Camel, you would be looking at writing hundreds of lines of code, searching the Internet for code snippets of how best to do reliable directory polling, parsing XML, using XPath libraries to help you split those XML fi les, setting up a JMS connection, and so forth. Camel hides all of that routine complexity into well-tested components so you just need to specify your integration problem as per the following example using the Spring XML DSL:

```
<route>
  <from uri="file://someDirectory?delay=60000"/>
  <split>
    <xpath>/xpath/to/record</xpath>
    <to uri="jms:queue:myProcessingQueue"/>
  </split>
</route>
```

Wow! We still remember when we first saw some Camel code, and were taken aback by how such a small amount of code could be so incredibly expressive.

This Camel example shows a **Route**, a definition (recipe) of a graph of channels to message processors, that says: consume files *from* `someDirectory` every 60,000 milliseconds; *split* that data based on an XPath *expression*; and send the resulting messages *to* a JMS queue named `myProcessingQueue`. That is exactly the problem we were asked to solve, and the Camel code effectively says just that. This not only makes it easy to create integration code, it makes it easy for others (including your future self) to look at this code and understand what it is doing.

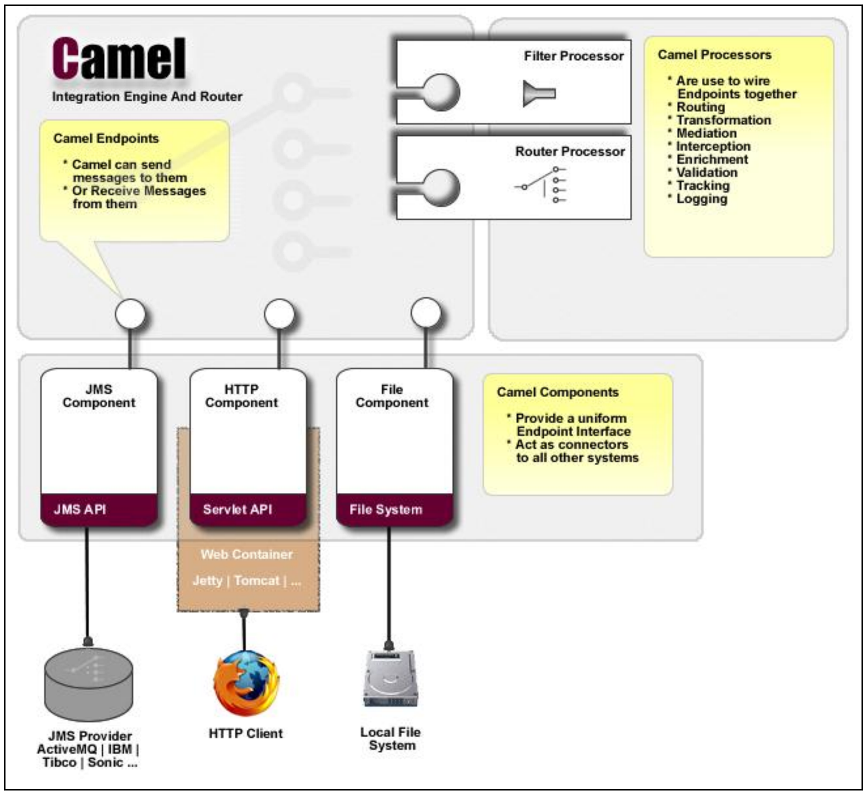
What is not obvious in this example is that this code also has default behaviors for handling errors (including retrying the processing of files), data type transformation such as File object to XML Document object, and connecting and packaging data to be sent on a JMS queue.

But what about when we need to use different endpoint technologies? How does Camel handle that? Very well, thank you very much. If our boss told us that now we need to pick up the files from a remote FTP server instead of a directory, we can make a simple change from the File Component to the FTP Component, and leave the rest of the integration logic alone:

```
<route>

<from uri="ftp://scott@remotehost/someDirectory?delay=60000"/>

<split>

<xpath>/xpath/to/record</xpath>

<to uri="jms:queue:myProcessingQueue"/>

</split>

</route>
```

This simple change from `file:` to `ftp:` tells Camel to switch from using the hundreds of lines of well-tested code doing local directory polling to the hundreds of lines of well-tested FTP directory polling code. Plus, your core record-processing logic to split and forward on to a JMS queue remains unchanged.

At the time of writing, there are over 160 components within the Camel community dealing with everything; from files, FTP, and JMS, to distributed registries such as Apache Zookeeper, low-level wire formats such as FIX and HL7, monitoring systems such as Nagios, and higher-level system abstractions that include Facebook, Twitter, SAP, and Salesforce. Many of these components were written by the team that created the technology you are trying to use, so they generally reflect best practices on using that technology. Camel allows you to leverage the best practices of hundreds of the best integration technologists in the world, all in an easy to use, open source framework.



Another big innovation with Camel is that it does not require the messages flowing through its processing channels (routes) to be of any fixed/canonical data type. Instead, Camel tracks the current data type of the message, and includes an extensible data type conversation capability that allows Camel to try to convert the message to the data type required by the next step in the processing chain. This also helps Camel in providing seamless integration with your existing Java libraries, as Camel can type convert to and from your Java methods that you call as part of your Camel route. This all combines into an extremely flexible mechanism where you can quickly and easily extend almost any part of Camel with some highly focused Java code.

There is so much more to Camel than what can be covered in this very brief overview.

Camel Concepts

A number of Camel architectural concepts are used throughout this book and are briefly explained here to provide a quick reference. Full details can be found on the Apache Camel website at `http://camel.apache.org`.

A Camel **Exchange** is a holder object that encapsulates the state of a conversation between systems. It contains properties, a variety of flags, a **message exchange pattern** or **MEP** (`InOnly` or `InOut`), and two messages (an `In` message and an `Out` message). Properties are expressed as a map of Strings to Objects, and are typically used by Camel and its components to store information pertaining to the processing of the exchange.

A **message** contains the payload to be processed by a processing step, as well as headers that are expressed as a map of Strings to Objects. You use headers to pass additional information about the message between processors. Headers are commonly used to override endpoint defaults.

An `In` message is always present on an exchange as it enters a processor. The processor may modify the `In` message or prepare a new payload and set it on the `Out` message. If a processor sets the `Out` message, the Camel context will move it to the `In` message of the exchange before handing it to the next processor. For more, see `http://camel.apache. org/exchange.html` and `http://camel.apache.org/message.html`.

A Camel **Processor** is the base interface for all message-processing steps. Processors include predefined EIPs such as a Splitter, calls to endpoints, or custom processors that you have created implementing the `org.apache.camel.Processor interface`. For more, see n`http://camel.apache.org/processor.html`.

A Camel **Route** is a series of message processing steps defined using Camel's DSL. A Route always starts with one consumer endpoint within a `from()` statement, and contains one or more processor steps. The processing steps within a route are loosely coupled, and do not invoke each other, relying on the Camel context instead to pass messages between them. For more, see `http://camel.apache.org/routes.html`.

The Camel **Context** is the engine that processes exchanges along the steps defined through routes. Messages are fed into a route based on a threading model appropriate to the component technologies being consumed from. Subsequent threading depends on the processors defined on the route.

A Camel **Component** is a library that encapsulates the communication with a transport or technology behind a common set of Camel interfaces. Camel uses these components to produce messages to or consume messages from those technologies. For a full list of components, see `http://camel.apache.org/components.html`.

A Camel **Endpoint** is an address that is interpreted by a component to identify a target resource, such as a directory, message queue, or database table that the component will consume messages from or send messages to. An endpoint used in a `from()` block is known as a Consumer endpoint, while an endpoint used in a `to()` block is known as a Producer endpoint. Endpoints are expressed as URIs, whose attributes are specific to their corresponding component. For more, see `http://camel.apache.org/endpoint.html`.

A Camel **Expression** is a way to script up Route in-line code that will operate on the message. For example, you can use the Groovy Expression Language to write inline Groovy code that can be evaluated on the in-flight message. Expressions are used within many EIPs to provide data to influence the routing of messages, such as providing the list of endpoints to route a message to as part of a Routing Slip EIP. For more, see `http://camel.apache.org/ expression.html`.

The Camel DSL

All integration routes are defined in Camel through its own domain-specific language (DSL). This book presents the two main DSL flavors when discussing routing, the Java DSL, and the Spring XML DSL. OSGi Blueprint XML DSL is modeled after Spring, and is touched on lightly in this book. The Spring and OSGi Blueprint XML DSLs are collectively referred to as the XML DSL). There are other DSL variants available for defining Camel routes, including Groovy and Scala. For details on these see the following links:

- Groovy DSL: `http://camel.apache.org/groovy-dsl.html`
- Scala DSL: `http://camel.apache.org/scala-dsl.html`

Here is an example of a classic Content Based Router configured in Camel using both the XML and Java DSLs. You can find out more details on this in the Content Based Router recipe in *Chapter 2, Message Routing*.

In the XML DSL, you would write the routing logic as:

```
<route>
  <from uri="direct:start"/>
  <choice>
    <when>
      <simple>${body} contains 'Camel'</simple>
      <log message="Camel ${body}"/>
    </when>
    <otherwise>
      <log message="Other ${body}"/>
    </otherwise>
  </choice>
  <log message="Message ${body}"/>
  </route>
```

In the Java DSL, the same route is expressed as:

```
from("direct:start")
  .choice()
    .when().simple("${body} contains 'Camel'")
      .log("Camel ${body}")
  .otherwise()
    .log("Other ${body}")
  .end()
  .log("Message ${body}");
```

The decision of which flavor of DSL to use is largely a personal one, and using one does not rule out using another alongside it. There are pros and cons to each, though none are functional. All of the DSL variants allow you to fully use Camel's features.

DSL	Pros	Cons
Java	▸ Routes can be defined in a very flexible manner (for example, the definition of a route can be conditional depending on the environment) ▸ `RouteBuilder` objects can be instantiated multiple times with different processors and endpoints, allowing route templating ▸ Routes tend to be shorter in terms of lines of code than the corresponding XML ▸ Each `RouteBuilder` can be tested independently without requiring the startup of every route in the whole Camel context	▸ Route definitions can sometimes get too clever in their use of advanced Java language features, such as large anonymous inner classes code blocks for in-lined processor steps, obscuring the intent of the route, and making future code maintenance more difficult ▸ As routes are defined using the builder pattern, automatic code reformatting in an IDE will mess up your indentation ▸ It may not always be obvious where a block of processing steps within an EIP ends ▸ It is sometimes necessary to break up the use of some EIPs within other EIPs to their own sub-routes due into limitations of using Java as a language for writing DSLs

DSL	Pros	Cons
XML	▸ Easily formatted automatically ▸ Easier to read by non-Java programmers ▸ Supported by round-trip engineering tools such as the Fuse IDE and hawtio ▸ Easier to use for environments where it may be difficult to deploy Java code, for example, alongside ActiveMQ configurations	▸ Verbose ("death by angle bracket") ▸ Not as easy to test as standalone Java routes; any Spring bean dependencies within routes pointing to external resources need to be mocked or stubbed in test Spring configurations

What This Book Covers

Chapter 1, *Structuring Routes*, introduces you to the fundamentals of structuring Camel integrations; getting the framework running inside Java and Spring applications, using Camel components, and breaking down and reusing routing logic.

Chapter 2, *Message Routing*, details the use of the main EIPs used to route messages within Camel integrations; everything from if-else style content-based routing to more complex, dynamic options.

Chapter 3, *Routing to Your Code*, describes how to interact with a Camel runtime from your Java code, and how your Java code can be used from within Camel routes.

Chapter 4, *Transformation*, provides some off-the-shelf strategies for converting between and manipulating common message formats such as Java objects, XML, JSON, and CSV.

Chapter 5, *Splitting and Aggregating*, takes a deep dive into the related Splitter and Aggregator EIPs. It details the impacts of completion conditions, parallel processing options, and using the EIPs in combination with each other.

Chapter 6, *Parallel Processing*, outlines Camel's support for scaling out processing through the use of thread pools, profiles, and asynchronous processors.

Chapter 7, *Error Handling and Compensation*, details the mechanisms provided by the Camel DSLs for dealing with failure, including capabilities for triggering compensating routing steps for non-transactional interactions that have already completed.

Chapter 8, *Transactions and Idempotency*, presents a number of variations for dealing with transactional resources (JDBC and JMS). It additionally details the handling of non-transactional resources (such as web services) in such a way that they will only ever be invoked once in the event of message replay or duplicates.

Chapter 9, *Testing*, outlines Camel's test support that allows you to verify your routes' behaviour without the need for backend systems. It also presents ways to manipulate routes with additional steps for testing purposes, without altering the code used at runtime.

Chapter 10, *Monitoring and Debugging*, describes Camel's support for logging, tracing, and debugging. Monitoring is examined through Camel's support for JMX, which includes the ability to define your own attributes and operations.

Chapter 11, *Security*, covers encrypting communication between systems, hiding sensitive configuration information, non-repudiation using certificates, and applying authentication and authorization to your routes.

Chapter 12, *Web Services*, shows you how to use Camel to invoke, act as a backend to, and proxy SOAP web services.

9
Testing

In this chapter, we will cover:

- Testing routes defined in Java
- Using mock endpoints to verify routing logic
- Replying from mock endpoints
- Testing routes defined in Spring
- Testing routes defined in OSGi Blueprint
- Auto-mocking of endpoints
- Validating route behavior under heavy load
- Unit testing processors and Bean Bindings
- Testing routes with fixed endpoints using AOP
- Testing routes with fixed endpoints using conditional events

Introduction

System integrations are traditionally a very difficult thing to test. Most commercial products, as well as home-cooked integrations, have no built-in support for automated testing. This usually results in verifying integration behavior through a manual approach involving triggering events/messages/requests from one window, and watching the end results in the affected systems. This approach has the following drawbacks:

- It is extremely time consuming, and inevitably leads to poor test coverage
- It is very difficult to test error conditions such as a system outage at just the wrong part of your integration flow

- It is complicated to verify the performance of integration code in isolation, as well as performing similar non-functional tests such as saturation testing
- It leaves no artifacts that can be used to detect regressions

Another fundamental problem with testing integrations using live backend systems is that it relies on the availability of those systems. This leaves your development workflow highly exposed to environment instability.

In most companies, the systems being integrated during development are often the test systems of other development teams. This means that they are likely to regularly be unavailable due to maintenance or code upgrades. The higher the number of systems being integrated, the greater the likelihood of any one of them is unavailable and prevents you from testing. This all-too-common scenario leaves teams scrambling for windows of opportunity when all their backends are up as the project deadline approaches.

Wouldn't it be nice if you could merely treat other systems as interfaces and test against those, rather than worrying about the physical implementations?

Camel comes with a built-in test kit that allows you to consider an integration to be just another set of code, in which backends are merely components that can be switched out for test versions.

This chapter covers the core elements of Camel that allow you to test your integrations, as well as providing some tips and techniques that should be applied given a set of circumstances.

If you have been looking at the example code from this book, you will have seen that the majority of the code is driven out of unit tests, and may have worked out which methods do what. This chapter and supporting samples focus exclusively on the JUnit framework, due to its overwhelming popularity among Java developers, being the de facto standard for unit testing. We will drill into the details of what is going on within Camel's support for JUnit so that you can then take that knowledge, and apply it to your testing framework of choice, regardless of whether Camel has explicit support for it or not.

The code for this chapter is contained within the `camel-cookbook-testing` module of the examples.

Testing routes defined in Java

This recipe will introduce you to the main parts of Camel's test support by showing you how to unit test a route defined within a `RouteBuilder` implementation. You will do this without relying on external systems to verify the interactions.

You will see how:

- The Camel framework gets set up and torn down

- Mock endpoints can be used to verify the message flow, through an **expect-run-verify** cycle, which should be familiar to those who have worked with mocking frameworks such as **EasyMock** (`http://www.easymock.org`) in the past
- Messages can be sent to endpoints from outside of Camel, allowing us to trigger routes with a range of payloads in order to verify the edge cases

Getting ready

To use Camel's core test support, you need to add a dependency for the `camel-test` library, which provides the support classes for JUnit testing as well as a transitive dependency on JUnit itself.

Add the following code to the dependencies section of your Maven POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test</artifactId>
  <version>${camel-version}</version>
  <scope>test</scope>
</dependency>
```

The Java code for this recipe is located in the `org.camelcookbook.examples.testing.java` package.

How to do it...

Let us test a simple route defined as follows:

```
public class SimpleTransformRouteBuilder extends RouteBuilder {
  @Override
  public void configure() throws Exception {
    from("direct:in")
      .transform(simple("Modified: ${body}"))
      .to("mock:out");
  }
}
```

The route consumes messages from an in-memory `direct:` endpoint, prepends the message body with the string `Modified:` and sends the result to a `mock:` endpoint.

To test this class, perform the following steps:

1. Create a test class that extends `org.apache.camel.test.junit4.CamelTestSupport`:

   ```
   public class SimpleTransformRouteBuilderTest
       extends CamelTestSupport {
     //...
   }
   ```

   The `CamelTestSupport` class is an abstract class that is responsible for instantiating the Camel context for the routes under test, creating utility objects, and injecting annotated properties into the test.

2. Override the `createRouteBuilder()` method, and instantiate your `RouteBuilder` implementation to be tested:

   ```
   @Override
   protected RouteBuilder createRouteBuilder()
       throws Exception {
     return new SimpleTransformRouteBuilder();
   }
   ```

3. Define the body of the test. Here, the `MockEndpoint` testing DSL is used to outline the messages that you expect an endpoint to receive. A test message is sent into the route via a `ProducerTemplate` instance provided for you by the `CamelTestSupport` base class. Finally, you assert that the expectations set on the mock endpoints in the Camel context were satisfied:

   ```
   @Test
   public void testPayloadIsTransformed()
       throws InterruptedException {
     MockEndpoint mockOut = getMockEndpoint("mock:out");
     mockOut.setExpectedMessageCount(1);
     mockOut.message(0).body().isEqualTo("Modified: Cheese");

     template.sendBody("direct:in", "Cheese");

     assertMockEndpointsSatisfied();
   }
   ```

   The `assertMockEndpointsSatisfied()` method is a catch-all helper method that checks that all mock endpoints within your route(s) are satisfied. You can verify that the expectations of individual mock endpoints have been met with `mockOut.assertIsSatisfied()`, where `mockOut` is replaced with a variable referencing your mock endpoint.

   If an assertion on a `MockEndpoint` fails, it will throw a `java.lang.AssertionError` in your test code when you check to see if it is satisfied.

As an alternative to explicitly fetching mocks, and referring to endpoints in each unit test, you can request an autoinjected `MockEndpoint` and `ProducerTemplate` by defining them as bean properties, and annotating them as follows:

```
@EndpointInject(uri = "mock:out")
private MockEndpoint mockOut;

@Produce(uri = "direct:in")
private ProducerTemplate in;
```

How it works...

The `CamelTestSupport` class is a convenience base class that prepares a Camel environment for your JUnit tests, without requiring that you repeatedly perform a number of repetitive steps every time you want to test a route.

As you can see from the preceding example, all that you need to do in order to set up a test is to override a base method `createRouteBuilder()`, and specify which properties you would like injected. Fundamentally, the base support class will perform the following:

- Start a Camel context before each test, adding any routes that you have specified through `createRouteBuilder()`, or `createRouteBuilders()` when testing multiple `RouteBuilder` implementations as part of a single test
- Inject any properties that you have annotated with `@Produce` and `@EndpointInject`
- Stop the Camel context after each test

If you want to reproduce the preceding behavior from **first principles**, that is without the support class, you first define a `CamelContext` variable as a private member of your test:

```
private CamelContext camelContext;
```

Before each test, instantiate the context, and initialize it with the `RouteBuilder` class under test:

```
@Before
public void setUpContext() throws Exception {
  this.camelContext = new DefaultCamelContext();
  camelContext.addRoutes(new SimpleTransformRouteBuilder());
  camelContext.start();
}
```

After each test method, shut down the Camel context:

```
@After
public void cleanUpContext() throws Exception {
  camelContext.stop();
}
```

You then need to access the Camel context directly, through your private variable, in order to obtain handles on mock endpoints and producer template:

```
MockEndpoint out =
    camelContext.getEndpoint("mock:out", MockEndpoint.class);
ProducerTemplate producerTemplate =
    camelContext.createProducerTemplate();
```

This first principles approach is useful when you are testing from a framework other than JUnit or TestNG (both of which are supported by Camel), or if for some reason you need to extend a base class that itself does not extend `CamelTestSupport`. This approach is not used that often, as extending Camel's test support classes has proven easier and has become a best practice.

There's more...

Testing routes that consume from a `direct:` endpoint and produce messages to `mock:` endpoints is easy. In real life though, your routes will be consuming from, and producing to, endpoint technologies such as CXF for SOAP services (see *Chapter 12*, *Web Services*), and ActiveMQ for messaging. How then do you go about testing that type of route?

The *Reusing routing logic through template routes* recipe in *Chapter 1*, *Structuring Routes* describes a technique for externalizing endpoints. This allows you to inject *real* endpoints when your route is deployed, and use `direct:` and `mock:` for testing when you instantiate your `RouteBuilder` in `createRouteBuilder()` method.

To do this, the previous `RouteBuilder` implementation should be modified as follows:

```
public class SimpleTransformDIRouteBuilder extends RouteBuilder {
  private String sourceUri;
  private String targetUri;

  // setters omitted

  @Override
  public void configure() throws Exception {
    from(sourceUri)
      .transform(simple("Modified: ${body}"))
      .to(targetUri);
  }
}
```

Then, all that is required to test the route is to inject the `direct:` and `mock:` endpoints inside the test class:

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
  SimpleTransformDIRouteBuilder routeBuilder =
    new SimpleTransformDIRouteBuilder();
  routeBuilder.setSourceUri("direct:in");
  routeBuilder.setTargetUri("mock:out");
  return routeBuilder;
}
```

Be careful when substituting endpoint technologies in this manner. Components may send in object types within exchanges that are different from those that you expected when writing your route. The unit testing of integrations should always be complemented with some integration testing using actual backend systems and their corresponding Camel Components.

There are further base methods within `CamelTestSupport` that you can override to define the details of how the Camel context will be set up. These are all prefixed with `create..()`.

You can override the `createCamelContext()` method to set up the Camel context in such a way that you can test with components that are not embedded in Camel's core library (`activemq:`, `cxf:`, `twitter:`, `leveldb:`, and so on). The following example configures the ActiveMQ Component for use in your tests:

```
@Override
public CamelContext createCamelContext() {
  CamelContext context = new DefaultCamelContext();

  ActiveMQComponent activeMQComponent = new ActiveMQComponent();
  activeMQComponent.setBrokerURL("vm:embeddedBroker");

  context.addComponent("activemq", activeMQComponent);

  return context;
}
```

See also

- Camel Test: `http://camel.apache.org/camel-test.html`
- Camel Mock Component: `http://camel.apache.org/mock.html`
- JUnit: `http://junit.org/`

Using mock endpoints to verify routing logic

The ability to verify message flow using mock endpoints was built into the Camel framework from its inception. The Mock Component in the `camel-core` library provides you with a testing DSL that allows you to verify which messages have reached various named `mock:` endpoints defined in your route. This recipe will describe how to make use of this mock DSL.

Getting ready

To use this recipe you should first have a route test set up as described in the *Testing routes defined in Java* recipe.

The Java code for this recipe is located in the `org.camelcookbook.examples.testing.mockreply` package.

How to do it...

To use mock endpoints, perform the following steps:

1. Within your route, use a `mock:` endpoint URI in any Camel DSL statement that produces a message to an endpoint, such as `to(..)` or `wireTap(..)`:

   ```
   from("direct:start")
     .choice()
       .when().simple("${body} contains 'Camel'")
         .setHeader("verified").constant(true)
         .to("mock:camel")
       .otherwise()
         .to("mock:other")
     .end();
   ```

2. Load the route into a Camel context, by overriding the `CamelTestSupport` class's `createRouteBuilder()` method, and start it as described in the *Testing routes defined in Java* recipe.
3. Obtain a `MockEndpoint` from the Camel context that corresponds to the `mock:` endpoint URI you want to verify against.

   If you are extending `CamelTestSupport`, you can obtain the endpoint as follows:

   ```
   MockEndpoint mockCamel = getMockEndpoint("mock:camel");
   ```

   If you are working from first principles, do the following:

   ```
   MockEndpoint mockCamel =
       camelContext.getEndpoint(
           "mock:camel", MockEndpoint.class);
   ```

4. Use the `MockEndpoint` to define the number and content of the messages that you expect to reach the endpoint once you exercise the route:

```
mockCamel.expectedMessageCount(1);
mockCamel.message(0).body().isEqualTo("Camel Rocks");
mockCamel.message(0).header("verified").isEqualTo(true);
```

5. Send messages into the route through a `ProducerTemplate`:

```
template.sendBody("direct:start", "Camel Rocks");
```

6. Verify that the expectations that you set on the mocks were met:

```
mockCamel.assertIsSatisfied();
```

[This general pattern of setting expectations on an injected mock object (endpoint in this case), testing the code, and checking the results is known as an **expect-run-verify** cycle.]

How it works...

A `MockEndpoint` is instantiated once when the Camel context is started, and collects all of the exchanges sent to it until the mock is destroyed. Any expectations defined on it are verified against that state.

[As `MockEndpoint` instances are stateful, it is essential that the Camel context be recreated between tests. If the Camel context is not recreated, then assertions against these mock endpoints may fail, as they still contain state from the previous test runs. For example, your number of messages expected assertion will fail as the mock will have gathered all of the messages from the last test run as well as the current run.]

The methods on a `MockEndpoint` can broadly be categorized as **expectations** or **assertions**:

- **Expectations** are defined *before* a mock is used (that is, before messages are sent to the route containing those mocks), and outline the expected state that the mock should accumulate by the end of the test.
- **Assertions** are evaluated *after* the mock has been used, and are used to verify that the expectations have been met. They are also used to evaluate conditions against the total set of the mock's accumulated state.

Expectation method names begin with `expect..()`, and aside from the examples already shown, include methods such as the following:

```
expectedBodiesReceived(Object...)
expectedBodiesReceivedInAnyOrder(Object...)
expectedFileExists(String)
expectedMessageMatches(Predicate)
expectedMinimumMessageCount(int)
```

The `message(int)` statement allows you to define expectations on individual messages, including evaluating expressions on any part of the exchange, as well as answering timing questions:

```
mock.message(0).simple("${header[verified]} == true");
mock.message(0).arrives().noLaterThan(50).millis().beforeNext();
```

Assertions allow you to verify that the expectations were met, as well as assessing the entire set of messages sent to an endpoint during a test. These methods are prefixed by `assert..()`, and include the following:

```
assertIsSatisfied()
assertIsSatisfied(int timeoutForEmptyEndpoints)
assertIsNotSafisfied()
assertMessagesAscending(Expression)
assertMessagesDescending(Expression)
assertNoDuplicates(Expression)
```

There are many more `assert` and `expect` methods available to you other than those covered here. Take a look at the `MockEndpoint` JavaDocs for more information.

There's more...

A `MockEndpoint` instance grants you access to the set of exchanges it received during a test run. This is useful when you want to compare `Exchange` objects received by the endpoint, or verify the mechanical behavior of a route by inspecting the internal state of an individual exchange.

The following code tests whether a particular header is the same for two `Exchange` objects:

```
List<Exchange> receivedExchanges = mock.getReceivedExchanges();
Exchange exchange0 = receivedExchanges.get(0);
Exchange exchange1 = receivedExchanges.get(1);
// JUnit assertion
assertEquals(exchange0.getIn().getHeader("verified"),
             exchange1.getIn().getHeader("verified"));
```

This mechanism is useful for testing such things as exchange equality, or inspecting graphs of objects associated with the message.

> You should only access received exchanges after calling `endpoint.assertIsSatisfied()`.

You can also combine the fetching of an individual exchange with the assertion that it was received through use of the `assertExchangeReceived(int)` helper method. The preceding code could be rewritten as:

```
Exchange exchange0 = mock.assertExchangeReceived(0);
Exchange exchange1 = mock.assertExchangeReceived(1);
assertEquals(exchange0.getIn().getHeader("verified"),
             exchange1.getIn().getHeader("verified"));
```

See also

- Camel Mock Component: `http://camel.apache.org/mock.html`
- The `ProducerTemplate` interface: `http://camel.apache.org/producertemplate.html`
- The `MockEndpoint` interface: `http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/component/mock/MockEndpoint.html`

Replying from mock endpoints

It is common that many of your integrations will call out to endpoints from which you will expect a reply, such as when invoking a web service. We saw how mock endpoints could be used to assess message content in a test in the *Using mock endpoints to verify routing logic* recipe. We will now use those same endpoints to define the test responses.

Getting ready

To use this recipe you should first have a route test set up as described in the *Testing routes defined in Java* recipe.

The Java code for this recipe is located in the `org.camelcookbook.examples.testing.mockreply` package.

How to do it...

In the following route, we have a mock endpoint from which we would like a response:

```
from("direct:in")
  .inOut("mock:replying")
  .to("mock:out");
```

To test this, get a handle to the mock endpoint that you want to simulate a reply from, as described in the *Testing routes defined in Java* recipe. Before a message is sent into the route, define how you want the endpoint to respond through the `whenAnyExchangeReceived (Processor)` method:

```
mockReplying.whenAnyExchangeReceived(new Processor() {
  @Override
  public void process(Exchange exchange) throws Exception {
    Message in = exchange.getIn();
    in.setBody("Hey " + in.getBody());
  }
});
```

How it works...

The `Processor` instance that you have provided to the mock endpoint will be called for every exchange that reaches that mock endpoint. This can be thought of as the default reply. On top of this, you can specify that you want a different processor to be engaged for a particular message. In the following example, we specify a different processor to be used for the first message received; all other messages received will use the processor provided in the previous call to `whenExchangeReceived`:

```
mockReplying.whenExchangeReceived(1, new Processor() {
  @Override
  public void process(Exchange exchange) throws Exception {
    Message in = exchange.getIn();
    in.setBody("Czesc " + in.getBody()); // Polish
  }
});
```

The number here is 1-indexed, where 1 indicates the first message to flow through the route. This can easily trip you up, as most people are in the habit of thinking of 0 as the first item.

Using a processor gives you maximum flexibility with what you can do when manipulating responses. It may be, however, that what you want to do can be described with a Camel Expression. In this case, the following approach might be more straightforward:

```
mockReplying.returnReplyBody(
    SimpleBuilder.simple("Hello ${body}"));
```

You can also set a header as the reply from an endpoint.

```
mockReplying.returnReplyHeader("someHeader",
    ConstantLanguage.constant("Hello"));
```

You cannot use the two `returnReply..()` methods on the same endpoint in the same test, as the second one will override the first. If you want to set a header *and* body, use the processor approach instead.

You may be used to having Expression Languages at your fingertips when writing a `RouteBuilder` implementations. Since test classes do not usually extend this base class, you have to use the static methods defined on the individual expression language classes directly. For example:

```
import static org.apache.camel.language.simple.
SimpleLanguage.simple;
```

See also

- Processor: `http://camel.apache.org/processor.html`
- Camel Mock Component: `http://camel.apache.org/mock.html`

Testing routes defined in Spring

This recipe expands on the core testing capabilities described so far by detailing the steps needed to test Camel routes defined using the XML DSL in a Spring application. You will learn how to assemble a test harness that replaces parts of the application in order to test your routes outside a deployment environment, including the substitution of Spring `${..}` placeholders with test values.

Getting ready

The Java code for this recipe is located in the `org.camelcookbook.examples.testing.spring` package. The Spring XML files are located under `src/main/resources/META-INF/spring`.

To use Camel's Spring test support, you need to add a dependency for the `camel-test-spring` library that provides the support classes for JUnit testing of Spring as well as a transitive dependency on JUnit itself.

Add the following to the `dependencies` section of your Maven POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test-spring</artifactId>
  <version>${camel-version}</version>
  <scope>test</scope>
</dependency>
```

How to do it...

Consider the following route, defined in the file located in `/META-INF/spring/simpleTransform-context.xml`:

```
<route>
  <from uri="direct:in"/>
  <transform>
    <simple>Modified: ${body}</simple>
  </transform>
  <log message="Set message to ${body}"/>
  <to uri="mock:out"/>
</route>
```

To test this route, perform the following steps:

1. Create a test class that extends `org.apache.camel.test.spring.CamelSpringTestSupport`:

   ```
   public class SimpleTransformSpringTest
       extends CamelSpringTestSupport {
     //...
   }
   ```

   The `CamelSpringTestSupport` class is an abstract class that is responsible for instantiating the Camel context with the routes under test, creating utility objects, and injecting annotated properties into the test.

2. Override the `createApplicationContext()` method from `CamelSpringTestSupport` and instantiate a Spring application context that loads the files containing the Camel routes under test:

```
@Override
protected AbstractApplicationContext
    createApplicationContext() {
  return new ClassPathXmlApplicationContext(
      "/META-INF/spring/simpleTransform-context.xml");
}
```

3. Define the body of the test. Use the `MockEndpoint` testing DSL to set expectations about the message you plan to receive during testing. A message is sent into the route via a `ProducerTemplate` instance provided for you by the `CamelTestSupport` class. Finally, assert that the expectations set on the mock endpoint were satisfied:

```
@Test
public void testPayloadIsTransformed()
    throws InterruptedException {
  MockEndpoint mockOut = getMockEndpoint("mock:out");
  mockOut.setExpectedMessageCount(1);
  mockOut.message(0).body().isEqualTo("Modified: Cheese");

  template.sendBody("direct:in", "Cheese");

  assertMockEndpointsSatisfied();
}
```

As an alternative to explicitly fetching mocks, and referring to endpoints in each unit test, you can request an autowired `MockEndpoint` and `ProducerTemplate`. These are defined as bean properties, and annotated as follows:

```
@EndpointInject(uri = "mock:out")
private MockEndpoint mockOut;

@Produce(uri = "direct:in")
private ProducerTemplate in;
```

How it works...

The `CamelSpringTestSupport` class is a convenience class that provides *feature-parity* with the `CamelTestSupport` class described in the *Testing routes defined in Java* recipe. It is responsible for performing the boilerplate work required to test Camel routes defined within Spring configuration files. At its most basic, the class will do the following:

- Start a Spring application defined by the context returned from `createApplicationContext()` *before each test*.
- Inject any properties that you have annotated with `@Produce`, `@EndpointInject`, or Spring's `@Autowired`. The last one allows your test code to get a handle on any object defined within the Spring application.
- Shut down the Spring application at the end of each test.

Feature-parity with `CamelTestSupport` means that aside from the implementation of a different base method to the Java testing example (`createApplicationContext()` versus `createRouteBuilder()` or `createRouteBuilders()`), `CamelSpringTestSupport` allows the test methods themselves to be written in exactly the same manner as their Java DSL equivalents. Both classes provide access to the same protected variables (`context` and `template`), and honor the same test lifecycle.

The testing method described has a drawback—it requires that a Camel base class is extended. This may not be something that you would like if you want to use another class as the base for your tests. To cater for this, Camel provides what is known as the **Enhanced Spring Test** option.

To make use of this, your test uses the JUnit 4 Runner functionality that allows a custom class to manage the lifecycle of a test. The rewritten test class appears as follows:

```
@RunWith(CamelSpringJUnit4ClassRunner.class)
@ContextConfiguration({
    "/META-INF/spring/simpleTransform-context.xml"})
public class SimpleTransformSpringRunnerTest {
  //...
}
```

As no base class is extended, your tests no longer have access to the protected variables, or utility methods such as `assertMockEndpointsSatisfied()` that are provided by `CamelSpringTestSupport`. The workaround for this is fairly straightforward. The Camel context is injected via the `@Autowired` annotation, just as any other Spring object:

```
@Autowired
private CamelContext camelContext;
```

When using Enhanced Spring Tests you invoke the static `MockEndpoint.assertIsSatisfied()` utility method to assert that all the mock endpoints in the Camel context interface were satisfied:

```
@Test
public void testPayloadIsTransformed()
    throws InterruptedException {
  mockOut.setExpectedMessageCount(1);
  mockOut.message(0).body().isEqualTo("Modified: Cheese");

  producerTemplate.sendBody("Cheese");

  MockEndpoint.assertIsSatisfied(camelContext);
}
```

One thing to watch out for is that the Enhanced Spring Tests have a different test lifecycle than those extending `CamelSpringTestSupport`. Instead of setting up and tearing down the entire Spring application between tests, *by default, the Spring application is set up once only at the start of the test class*. This impacts the mock endpoints in classes with more than one test method, as these are not reset to their initial state between tests.

To get around this, add the following class annotation to the test:

```
@DirtiesContext(
    classMode=ClassMode.AFTER_EACH_TEST_METHOD)
```

There's more...

Spring applications are usually broken up into multiple context files that address a particular fragment of an application each—you would generally see one file per horizontal application tier, or per vertical stack, of functionality. By making use of this approach, you can substitute alternative Spring configuration files when assembling your tests to replace portions of an application. This allows you to stub out objects used in your routes that may be difficult to wire up in a test environment.

It is usually a good idea to keep your `PropertyPlaceholderConfigurer` configuration in a different Spring XML file from your routes. This allows you to plug in a test version of those properties for testing.

We can use this particular mechanism to fully externalize our route endpoints, and use dependency injection to make the route more easily testable. Consider the following example:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <propertyPlaceholder id="properties"
                       location="ref:ctx.properties"/>
  <route>
    <from uri="{{start.endpoint}}"/>
    <transform>
      <simple>Modified: ${body}</simple>
    </transform>
    <log message="Set message to ${body}"/>
    <to uri="{{end.endpoint}}"/>
  </route>
</camelContext>
```

By providing an alternative `PropertyPlaceholderConfigurer` instance–defined as a `bean` with an `id` of `ctx.properties`–in a test file in a test file, we can exercise the route using `direct:` and `mock:` endpoints, while referring to the actual technology endpoints in the production configuration.

Using this approach of substituting in test endpoints, you are validating the logic of the route, and not the behavior of the route with the actual components. This may give you a false sense of security, as the routing logic may not work in the same way when the real endpoints are in use. For example, a consumer endpoint may use a different type of object in the exchange body than what your substituted test endpoint provides, and this will subtlety change your logic.

This sort of testing should always be used in conjunction with an integration test that exercises the real endpoints.

The following example uses the Spring `util` and `context` namespaces for brevity to substitute test endpoints into the preceding route:

```
<util:properties id="ctx.properties"
                 location="classpath:spring/test.properties"/>
<context:property-placeholder properties-ref="ctx.properties" />
```

The `test.properties` file contains the test versions of the properties:

```
start.endpoint=direct:in
end.endpoint=mock:out
```

Splitting out the `PropertyPlaceholderConfigurer` configuration into a different file is particularly useful if the file containing your Camel context definition also contains other beans that need properties injected.

It is a good practice to place any beans that have complex dependencies, or could be considered services, in a separate configuration file to your routes. This makes it easy to provide an alternative version of those beans in your test by providing a different version of that file to the test runtime.

You can alternatively feed properties directly into the context through the `Properties` component by overriding the following method from `CamelTestSupport`:

```
@Override
protected Properties
    useOverridePropertiesWithPropertiesComponent() {
  Properties properties = new Properties();
  properties.setProperty("start.endpoint", "direct:in");
  properties.setProperty("end.endpoint", "mock:out");
  return properties;
}
```

If your `propertyPlaceholder` refers to properties files that are not discoverable at test time, you can instruct the framework to ignore the error instead of throwing an exception that would prevent the Camel context from starting up. You do this by overriding the following method in your test:

```
@Override
protected Boolean ignoreMissingLocationWithPropertiesComponent() {
  return true;

}
```

See also

- Spring Testing: `http://camel.apache.org/spring-testing.html`
- Properties Component: `http://camel.apache.org/properties.html`
- JUnit: `http://junit.org/`
- The *Auto-mocking of endpoints* recipe
- The *Testing routes with fixed endpoints using AOP* recipe
- The *Testing routes with fixed endpoints using conditional events* recipe

Testing routes defined in OSGi Blueprint

This recipe expands on the core testing capabilities described so far by detailing the steps needed to test Camel routes defined using the XML DSL in an OSGi Blueprint application. You will learn how to assemble a test harness that replaces parts of the application in order to test your routes outside an OSGi deployment environment, including the substitution of OSGI Configuration Admin service `${..}` placeholders with test values. Camel allows Blueprint-based routes to be tested outside of an OSGI container, through a project called PojoSR that makes it possible to test the code without deploying it.

Getting ready

To use Camel's Blueprint test support, you need to add a dependency for the `camel-test-blueprint` library that provides the support classes for JUnit testing of Blueprint as well as a transitive dependency on JUnit itself.

Add the following to the `dependencies` section of your Maven POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test-blueprint</artifactId>
  <version>${camel-version}</version>
  <scope>test</scope>
</dependency>
```

The Java code for this recipe is located in the `org.camelcookbook.examples.testing.blueprint` package. The Blueprint XML files are located under `src/main/resources/OSGI-INF/blueprint`.

How to do it...

Consider the following route, defined in `simpleTransform-context.xml`:

```
<blueprint
    xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.osgi.org/xmlns/blueprint/v1.0.0
        http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
      http://camel.apache.org/schema/blueprint
        http://camel.apache.org/schema/blueprint/
        camel-blueprint.xsd">
  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route>
```

```
        <from uri="direct:in"/>
        <transform>
          <simple>{{transform.message}}: ${body}</simple>
        </transform>
        <to uri="mock:out"/>
      </route>
    </camelContext>
  </blueprint>
```

Here, we use a placeholder to fetch the transform message. We define an OSGi Configuration Admin property placeholder in a file alongside it, located in `simpleTransform-properties-context.xml` (namespace declarations partially shown):

```
<blueprint ...
    xmlns:cm="http://aries.apache.org/blueprint/xmlns/
              blueprint-cm/v1.0.0"
    xsi:schemaLocation="...
      http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0
        http://aries.apache.org/schemas/blueprint-cm/
        blueprint-cm-1.0.0.xsd">
  <cm:property-placeholder
      persistent-id="org.camelcookbook.testing">
    <cm:default-properties>
      <cm:property name="transform.message"
                   value="Modified"/>
    </cm:default-properties>
  </cm:property-placeholder>
</blueprint>
```

It is a good practice to place any beans that have complex dependencies, or could be considered services, in a separate configuration file from your routes. This makes it easy to provide an alternative version of those beans in your test by providing a different version of that file to the test runtime.

To test this route, perform the following steps:

1. Create a test class that extends `org.apache.camel.test.blueprint.CamelBlueprintTestSupport`:

   ```
   public class SimpleTransformBlueprintTest
       extends CamelBlueprintTestSupport {
     //...
   }
   ```

The `CamelBlueprintTestSupport` class is an abstract class that is responsible for instantiating the Camel context with the routes under test, creating utility objects, and setting up a simulated OSGi environment. To test a route, override the `getBlueprintDescriptor()` method, and return the Blueprint configuration files that contain the Camel routes under test, as well as any support beans, as a *comma-separated string*:

```
@Override
protected String getBlueprintDescriptor() {
  return "/OSGI-INF/blueprint/simpleTransform-context.xml,"
       + "/OSGI-INF/blueprint/simpleTransform-props-context.xml";
}
```

2. Define the body of the test. Here, the `MockEndpoint` testing DSL is used to establish the messages that you expect an endpoint to receive. A message is sent into the route via the `ProducerTemplate` instance provided for you by the `CamelTestSupport` base class. Finally, you assert that the expectations set on the mock endpoint were satisfied:

```
@Test
public void testPayloadIsTransformed()
    throws InterruptedException {
  MockEndpoint mockOut = getMockEndpoint("mock:out");
  mockOut.setExpectedMessageCount(1);
  mockOut.message(0).body().isEqualTo("Modified: Cheese");

  template.sendBody("Cheese");

  assertMockEndpointsSatisfied();
}
```

As an alternative to explicitly fetching mocks, and referring to endpoints in each unit test, you can request an injected `MockEndpoint` and `ProducerTemplate` by defining them as fields and annotating them as follows:

```
@EndpointInject(uri = "mock:out")
private MockEndpoint mockOut;

@Produce(uri = "direct:in")
private ProducerTemplate in;
```

How it works...

Camel's Blueprint test support avoids the need for an OSGi runtime by instantiating the Blueprint context files using the PojoSR library. This library provides a container that supports an OSGi-style service registry without a full-blown OSGi framework. This leads to the container starting and stopping faster, which is more appropriate for unit testing.

The `CamelBlueprintTestSupport` class is a convenience class that provides *feature-parity* with the `CamelTestSupport` class described in the *Testing routes defined in Java* recipe. It is responsible for performing the boilerplate work required to test Camel routes defined within Blueprint configuration files. At its most basic, the class will do the following:

- Start a Blueprint application defined by the Blueprint descriptors returned from `getBlueprintDescriptor()` *before each test*
- Inject any properties that you have annotated with `@Produce` and `@EndpointInject`
- Shut down the Blueprint application at the end of each test

Feature-parity with `CamelTestSupport` means that aside from the implementation of different base methods to the Java testing example (`getBlueprintDescriptor()` versus `createRouteBuilder()` or `createRouteBuilders()`), `CamelBlueprintTestSupport` allows the test methods themselves to be written in exactly the same manner to their Java DSL equivalents. Both classes provide access to the same protected variables (`context` and `template`), and honor the same test lifecycle.

At the time of writing this book the `camel-test-blueprint` library is not as functionally mature as `camel-test-spring`. The only test framework that is supported is JUnit 4. There is also no alternative that would enable you to work with other test frameworks as per the Extended Spring Test mechanism described in the *Testing routes defined in Spring* recipe.

There's more...

The Blueprint Configuration Admin support allows you to define default values for properties. These values are associated with a **Persistent ID** (**pid**) that corresponds to a named map, which is typically overridden inside an OSGi environment. When testing, you will typically want to override these values with something more suited to your needs.

A simple solution, if your `<cm:property-placeholder/>` block is defined in a separate file from your routes, might be to define a test version of that file and return it from `getBlueprintDescriptor()`.

A better approach is to override the properties individually by overriding the following method in your test class:

```
@Override
protected String useOverridePropertiesWithConfigAdmin(
    Dictionary props) throws Exception {
  props.put("transform.message", "Overridden");
  return "org.camelcookbook.testing"; // Persistent ID
}
```

If you have a lot of properties that require overriding, or a lot of tests that would require you to repeat this code block, you can provide the location of a properties file (ending in `.properties` or `.cfg`) and the Persistent ID to override as follows:

```
@Override
protected String[] loadConfigAdminConfigurationFile() {
  return new String[] {
    "src/test/resources/blueprint/testProperties.cfg",
    "org.camelcookbook.testing" // Persistent ID
  };
}
```

The Blueprint test mechanism has been written with Maven in mind, and loads the properties file relative to the project root. The test as specified previously will most likely not run from within your IDE, as the PojoSR container will execute within a different absolute directory.

When this mechanism is combined with `useOverridePropertiesWithConfigAdmin()`, the properties file will override the default properties provided in the `<cm:property-placeholder/>` block, and the manually set properties will in turn override the values in the file.

We can use this particular mechanism to make testing easier. Consider the following example:

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  <route>
    <from uri="{{in.endpoint}}"/>
    <transform>
      <simple>{{transform.message}}: ${body}</simple>
    </transform>
    <to uri="{{out.endpoint}}"/>
  </route>
</camelContext>
```

By externalizing the endpoints we can now exercise the route in a test using `direct:` and `mock:`, while referring to the actual technology endpoints in the production configuration.

See also

- Blueprint Testing: `http://camel.apache.org/blueprint-testing.html`
- Properties Component: `http://camel.apache.org/properties.html`
- PojoSR: `http://code.google.com/p/pojosr/`
- The *Auto-mocking of endpoints* recipe
- The *Testing routes with fixed endpoints using AOP* recipe
- The *Testing routes with fixed endpoints using conditional events* recipe

Auto-mocking of endpoints

In the recipes that have involved mock testing so far, it has been necessary to provide a `mock:` endpoint URI directly into a route, either explicitly or through dependency injection. Camel's test support classes provide a mechanism for automatically mocking endpoints that allow you to more easily test routes with embedded URIs. This recipe will show you how to exercise this functionality.

Getting ready

The Java code for this recipe is located in the `org.camelcookbook.examples.testing.automocking` package. The Spring route used here is located under `src/main/resources/META-INF/spring/fixedEndpoints-context.xml`.

How to do it...

Consider the following route, with fixed endpoints:

```
<from uri="activemq:in"/>
<transform>
  <simple>Modified: ${body}</simple>
</transform>
<log message="Set message to ${body}"/>
<to uri="activemq:out"/>
```

In order to set up a mock endpoint on the `to(..)` node without changing the route, perform the following steps:

1. Create a test that extends `CamelSpringTestSupport` as described in the *Testing routes defined in Spring* recipe, loading the preceding XML configuration. Then, override the `isMockEndpoints()` method, returning the name (endpoint URI) of the endpoint(s) that you would like to override:

   ```
   @Override
   public String isMockEndpoints() {
     return "activemq:out";
   }
   ```

   If using Enhanced Spring Testing, your test class should be annotated with:

   ```
   @MockEndpoints("activemq:out")
   ```

2. A mock endpoint can now be fetched from the Camel context, through the mocked endpoint URI, prefixed with `mock:`

   ```
   @EndpointInject(uri="mock:activemq:out")
   MockEndpoint mockOut;
   ```

   It is now possible to test the route as usual, by sending in a message to the start endpoint, and asserting expectations on the `MockEndpoint` interface.

How it works...

Camel will create a mock endpoint for the URIs that you have specified, and use the full name of the original URI, including the scheme (`activemq` in the preceding example), as the name portion of the `mock:` endpoint URI.

As such, `activemq:out` becomes `mock:activemq:out`.

Auto-mocking does not replace the existing endpoints. A message will still be sent to `activemq:out` in the previous example, as well as `mock:activemq:out`. To skip sending to the original endpoint, you should override the `isMockEndpointsAndSkip()` method instead of `isMockEndpoints()`.

The String describing which endpoints to mock may be a full endpoint URI, a wildcard (`*`), or a regular expression. It may also contain `{{..}}` property placeholders, which will be resolved first, before the remainder of the expression is processed.

One thing to note is that if the endpoint URI to be mocked contains attributes, then the matching String used in `isMockEndpoints()` or `@MockEndpoints` needs to use a wildcard in order for the framework to identify a match against the URI:

Considering the original endpoint URI:

```
activemq:out?timeToLive=10000
```

The following wildcard will match that as an endpoint to be mocked:

```
activemq:out?*
```

The URI that is then used to fetch the mock endpoint from the Camel context will not contain any of the original endpoint URI attributes

```
mock:activemq:out
```

There's more...

If you want to mock multiple endpoints within your route, you can specify the list as a regular expression:

```
(activemq:first|activemq:second)
```

You will then be able to access these as usual:

```
@EndpointInject(uri="mock:activemq:first")
@EndpointInject(uri="mock:activemq:second")
```

The approach of overriding `isMockEndpoints()` applies to not only to testing Spring-based routes, but those defined in OSGi Blueprint, and Java `RouteBuilder` implementations as well. The method is defined on `CamelTestSupport`, which both `CamelSpringTestSupport` and `CamelBlueprintTestSupport` extend.

Validating route behavior under heavy load

Every so often you will find yourself developing routes that you need to validate for performance under load, as well as their general correctness. The mocking techniques that have been discussed so far will only help you determine whether a test message correctly exercises the route.

This recipe will show how you can use the `DataSet` Component to generate a set of test data and play it through your route. It will then demonstrate how this same component can be used as a bulk mocking mechanism to validate your route logic under load. This technique is not a replacement for proper system integration load testing. It provides a means to more easily unit test scenarios involving larger numbers of messages.

Consider the following routing logic, which is defined in a `RouteBuilder` implementation that includes property setters allowing us to inject start and end URIs:

```
public class SlowlyTransformingRouteBuilder
    extends RouteBuilder {
  private String sourceUri; // setters omitted...
  private String targetUri;

  @Override
  public void configure() throws Exception {
    from(sourceUri)
      .to("seda:transformBody");

    from("seda:transformBody?concurrentConsumers=15")
      .transform(simple("Modified: ${body}"))
      .delay(100) // pretend this is a slow transformation
      .to("seda:sendTransformed");

    from("seda:sendTransformed") // one thread used here
      .resequence().simple("${header.mySequenceId}").stream()
      .to(targetUri);
  }
}
```

This `RouteBuilder` implementation uses the SEDA Component to allow multiple threads to execute a slow transformation on a number of incoming messages in parallel. When the messages are successfully transformed, they are handed to another route that uses the **Resequencer** EIP. The Resequencer is a pattern built into Camel that sorts exchanges flowing through a route. In this instance, exchanges are sorted according to a header that identifies the original order that they were sent in (`mySequenceId`). This header is set on the incoming messages before they enter the first route.

We would like to verify that the messages are sent out in the order that they came in, and also that we have enough `concurrentConsumers` set on the second route to deal with 100 messages a second.

Getting ready

The Java code for this recipe is located in the `org.camelcookbook.examples.testing.dataset` package.

How to do it...

To apply the DataSet Component for bulk testing, perform the following steps:

1. To generate a set of inbound test messages, extend the `DataSetSupport` class, overriding the abstract `createMessageBody()` method, as well as the `applyHeaders()` template method:

   ```
   public class InputDataSet extends DataSetSupport {
     @Override
     protected Object createMessageBody(long messageIndex) {
       return "message " + messageIndex;
     }

     protected void applyHeaders(Exchange exchange,
                                 long messageIndex) {
       exchange.getIn()
           .setHeader("mySequenceId", messageIndex);
     }
   }
   ```

   The `DataSetSupport` class is an abstract implementation of the `DataSet` interface, which is used by the `DataSet` Component to generate messages.

2. To generate the messages that will be expected at the end of the route, repeat by extending `DataSetSupport` again, this time skipping the setting of headers:

   ```
   public class ExpectedOutputDataSet extends DataSetSupport {
     @Override
     protected Object createMessageBody(long messageIndex) {
       return "Modified: message " + messageIndex;
     }
   }
   ```

3. In your `CamelTestSupport` test class, register the two beans with the Camel context:

   ```
   @Override
   protected CamelContext createCamelContext()
       throws Exception {
     final int testBatchSize = 1000;
     InputDataSet inputDataSet = new InputDataSet();
     inputDataSet.setSize(testBatchSize);

     ExpectedOutputDataSet expectedOutputDataSet =
   ```

```
      new ExpectedOutputDataSet();
  expectedOutputDataSet.setSize(testBatchSize);

  SimpleRegistry registry = new SimpleRegistry();
  registry.put("input", inputDataSet);
  registry.put("expectedOutput", expectedOutputDataSet);

  return new DefaultCamelContext(registry);
}
```

If using this style of testing with Spring, the two `DataSet` implementations should be registered as beans in the Spring context.

4. Instantiate your `RouteBuilder` and set two `dataset:` endpoints: the `startURI` endpoint pointing to the `InputDataSet` bean, and the `targetUri` endpoint pointing to the `ExpectedOutputDataSet`:

```
@Override
protected RouteBuilder createRouteBuilder()
    throws Exception {
  SlowlyTransformingRouteBuilder routeBuilder =
      new SlowlyTransformingRouteBuilder();
  routeBuilder.setSourceUri(
      "dataset:input?produceDelay=-1");
  routeBuilder.setTargetUri("dataset:expectedOutput");
  return routeBuilder;
}
```

The `produceDelay` option is set to `-1` in the source URI so as to start sending messages immediately.

In your test method, fetch the `dataset:expectedOutput` mock endpoint, set the maximum time that it should wait to receive its expected 1,000 messages, and assert that it was satisfied:

```
@Test
public void testPayloadsTransformedInExpectedTime()
    throws InterruptedException {
  MockEndpoint expectedOutput =
      getMockEndpoint("dataset:expectedOutput");
  expectedOutput.setResultWaitTime(10000); // 10 seconds
  expectedOutput.assertIsSatisfied();
}
```

How it works...

The DataSet Component is used here both as a consumer endpoint to generate test messages, and as a producer endpoint to act as a bulk mock testing mechanism.

When the Camel context starts the route, a single thread will be created by the `dataset:input` endpoint. This thread will perform the following steps in a loop:

- Generate the message body by calling `createMessageBody()` on the `input` bean
- Set the message headers through `applyHeaders()`
- Process the resulting exchange through the route

Each time a message is constructed, a `messageIndex` parameter is passed in that allows your code to vary the message.

A `produceDelay` attribute on the URI allows you to set how long the thread should sleep between having finished the routing of the message and generating the next message; `-1` indicates no delay.

When used as a producer in a `to(..)` block statement, a `DataSetEndpoint` interface acts as a bulk mock endpoint that automatically sets its own expectations.

At the other end of our routes, `dataset:expectedOutput` waits to receive 1,000 messages. If the time set in your test by `setResultWaitTime()` expires, the assertion will fail. If the messages were received, the endpoint will generate that same number of exchanges through its own implementation class. It does this in order to compare the received messages against the expected ones.

The two sets of `Exchange` objects are compared to each other *in order*. The ordering is determined by matching the `CamelDataSetIndex` attributes that are set by the consumer and producer `dataset:` endpoints. The two message bodies are then compared for equality. Other headers and attributes are not considered. A mismatch of either the index attributes, or the bodies will result in the assertion failing.

There's more...

It is possible to generate a default set of message headers by overriding `DataSetSupport.populateDefaultHeaders(Map<String, Object>)`. These will be overwritten on a per-message basis in `applyHeaders()`.

There is a `consumeDelay` attribute that can also be used as part of the producer URI to simulate a slow consumer.

See also

- DataSet Component: `http://camel.apache.org/dataset.html`
- Resequencer: `http://camel.apache.org/resequencer.html`

Unit testing processors and Bean Bindings

When developing complex `Processor` implementations, it is useful to test them in isolation to ensure that they are fully exercised—something that may not necessarily be straightforward in a production route. Likewise, when developing Java classes marked with Camel annotations for bean binding, you want to check the binding logic as well as logic contained within the class. This recipe presents an approach for testing these types of scenarios.

Getting ready

The Java code for this recipe is located in the `org.camelcookbook.examples.testing.exchange` package.

How to do it...

Processors are typically used for composite actions that involve modifying the body of an exchange as well as a number of headers. Here is a `Processor` implementations `process()` method that we would like to test:

```
@Override
public void process(Exchange exchange) throws Exception {
  final String something = "SOMETHING";
  Message in = exchange.getIn();
  String action = in.getHeader("action", String.class);
  if ((action == null) || (action.isEmpty())) {
    in.setHeader("actionTaken", false);
  } else {
    in.setHeader("actionTaken", true);
    String body = in.getBody(String.class);
    if (action.equals("append")) {
    in.setBody(body + " " + something);
    } else if (action.equals("prepend")) {
      in.setBody(something + " " + body);
    } else {
      throw new IllegalArgumentException(
          "Unrecognized action requested: [" + action + "]");
    }
  }
}
```

In order to test this processor with an Exchange instance, perform the following steps:

1. Extend `CamelTestSupport` and define an inline route through a `RouteBuilder` implementation that contains only that processor:

```
@Override
protected RouteBuilder createRouteBuilder()
    throws Exception {
  return new RouteBuilder() {
    @Override
    public void configure() throws Exception {
      from("direct:in")
        .process(new ComplicatedProcessor())
        .to("mock:out");
    }
  };
}
```

2. Test the route as usual:

```
@Test
public void testPrepend() throws Exception {
  MockEndpoint mockOut = getMockEndpoint("mock:out");
  mockOut.message(0).body().isEqualTo("SOMETHING text");
  mockOut.message(0).header("actionTaken").isEqualTo(true);

  Map<String, Object> headers =
      new HashMap<String, Object>();
  headers.put("action", "prepend");

  template.sendBodyAndHeaders("direct:in",
                              "text", headers);

  assertMockEndpointsSatisfied();
}
```

How it works...

By developing a throwaway route, we have enabled the processor under test to be executed inside an actual Camel context. This allows us to feed sample messages to the processor inside the route using a `ProducerTemplate` instance, and check the results using a mock endpoint. This approach distances us from the details of constructing an Exchange object by hand, and instead uses Camel to do it for us.

There's more...

Camel allows you to invoke bean methods directly from your route via a `bean(..)` statement, without needing that you refer to any Camel APIs in the method being invoked. A mechanism called **Bean Binding** is used to map the contents of the exchange to your method parameters (see the *Routing messages directly to a Java method* recipe in *Chapter 3*, *Routing to Your Code*). Using this mechanism you can access any part of the exchange without relying on the Camel API directly by using Camel's runtime annotations on the method arguments. While the resulting POJO can be tested like any other object, in order to test the bindings, you need to instantiate the bean inside a route and test as done before.

See also

- Processor: `http://camel.apache.org/processor.html`
- Bean Binding: `http://camel.apache.org/bean-binding.html`
- The *Routing messages directly to a Java method* recipe in *Chapter 3*, *Routing to Your Code*

Testing routes with fixed endpoints using AOP

When working with Camel you may at some point need to test a route that has endpoint URIs hardcoded into it. This tends to be more typical when testing Spring or Blueprint routes.

Consider the following route defined in `/META-INF/spring/fixedEndpoints-context.xml`:

```
<route id="modifyPayloadBetweenQueues">
  <from uri="activemq:in"/>
  <transform>
    <simple>Modified: ${body}</simple>
  </transform>
  <to uri="activemq:out"/>
</route>
```

The route endpoints here make use of the ActiveMQ Component to consume from one queue and publish to another. We would like to test this logic as described in the *Testing routes defined in Spring* recipe, in a pure unit test without making any changes to the route.

Camel provides you with a built-in form of **Aspect-Oriented Programming** (**AOP**) in the form of the `adviceWith(..)` Java DSL. This feature allows you to modify routing logic, once it has been loaded into the Camel context, to replace the route endpoints with `direct:` and `mock:` in order to make it easier to test. This recipe will show you how to use this approach to modify an existing route with fixed endpoints.

Getting ready

The Java code for this recipe is located in the `org.camelcookbook.examples.testing.advicewith` package.

How to do it...

In order to apply AOP to your tests, perform the following steps:

1. In a test class that extends `CamelSpringTestSupport`, override the following method:

   ```
   @Override
   public boolean isUseAdviceWith() {
     return true;
   }
   ```

   This tells the base class to not start the Camel context before running each test method, as it would otherwise do.

2. In your test method, fetch the route by its `id` (`modifyPayloadBetweenQueues`), and use the `adviceWith(..)` block to apply a new set of routing instructions over it using an `AdviceWithRouteBuilder`:

   ```
   context.getRouteDefinition("modifyPayloadBetweenQueues")
       .adviceWith(context, new AdviceWithRouteBuilder() () {
     @Override
     public void configure() throws Exception {
       replaceFromWith("direct:in");

       interceptSendToEndpoint("activemq:out")
         .skipSendToOriginalEndpoint()
         .to("mock:out");
     }
   });
   ```

3. Start the Camel context manually, and proceed to test the route as you would otherwise do with `direct:` and `mock:` endpoints:

    ```
    context.start();

    MockEndpoint out = getMockEndpoint("mock:out");
    out.setExpectedMessageCount(1);
    out.message(0).body().isEqualTo("Modified: Cheese");

    template.sendBody("direct:in", "Cheese");

    assertMockEndpointsSatisfied();
    ```

How it works...

The `adviceWith(..)` statement allows us to provide a set of instructions to lay over the top of a route in order to modify its behavior. We define these instructions through a specialized DSL provided to us by overriding the `AdviceWithRouteBuilder` abstract class.

The `isUseAdviceWith()` method is defined in `CamelTestSupport`, which both `CamelSpringTestSupport` and `CamelBlueprintTestSupport` extend. This means that it can be used regardless of whether you are testing Java, Spring, or OSGi Blueprint routes. To achieve the same behavior in an Extended Spring Test, apply the `org.apache.camel.test.spring.UseAdviceWith` annotation to the test class to indicate the same thing.

There's more...

The `adviceWith(..)` mechanism can be used for a lot more than replacing the consumer and producer endpoints on a route. Using a technique called **weaving**, you can manipulate the route itself to remove or replace individual nodes. This can be handy if one step of your route is responsible for substantial processing that you want to skip for the purposes of testing, as in the following route:

```
from("direct:in").id("slowRoute")
  .process(new ReallySlowProcessor()).id("reallySlowProcessor")
  .to("mock:out");
```

To bypass the long-running process for testing, you can use the following to replace it with a much faster substitute:

```
context.getRouteDefinition("slowRoute")
  .adviceWith(context, new AdviceWithRouteBuilder() {
    @Override
    public void configure() throws Exception {
      weaveById("reallySlowProcessor").replace()
```

```
            .transform().simple("Fast reply to: ${body}");
        }
      });
```

In the preceding example, the node is fetched through `weaveById(String)`, where the `id` attribute is one that you have assigned to the `process` DSL statement. You can also fetch nodes by a regular expression that matches the `toString()` representation of the node via `weaveByToString(String)`, or by the internal Camel type of the node (for example, `ToDefinition.class`) through `weaveByType(class)`.

Once a node has been selected via a `weave..()` method, the DSL gives you the following additional modification options:

- `before()`: Add the nodes that follow immediately before the selected node
- `after()`: Add the nodes that follow after the selected node, but before the next one in the original route
- `replace()`: Replace the selected node with the following node(s)
- `remove()`: Eliminate the selected node from the route
- The methods `weaveAddFirst()` and `weaveAddLast()` can be used to add nodes to the start or end of a route that can be useful when you want to validate inputs or outputs, especially in routes used from other routes

See also

- AdviceWith: `http://camel.apache.org/advicewith.html`

Testing routes with fixed endpoints using conditional events

When working with Camel, you may at some point need to test a route that has endpoint URIs hardcoded into it. This tends to be more typical when testing Spring or Blueprint routes.

Consider the following route defined in `/META-INF/spring/fixedEndpoints-context.xml`:

```
<route id="modifyPayloadBetweenQueues">
  <from uri="activemq:in"/>
  <transform>
    <simple>Modified: ${body}</simple>
  </transform>
  <to uri="activemq:out"/>
</route>
```

The route endpoints here make use of the ActiveMQ Component to consume from one queue and publish to another. We would like to test this logic as described in the *Testing routes defined in Spring* recipe, in a pure unit test without making any changes to the route.

Camel provides you with a notification-based DSL for testing this type of route via the `NotifyBuilder`. Unlike the `adviceWith(..)` DSL described in the *Testing routes with fixed endpoints using AOP* recipe, this approach does not rely on modifying the running route in any way. This is useful when you want to engage the actual endpoint technologies in the test, for example when validating performance, while still validating the routing logic.

As such, `NotifyBuilder` can be considered as a form of **black-box** testing where you validate that the outputs match the given inputs without needing to know anything about the internals of the routing. This is in contrast to the testing approaches that we have seen previously that adopt a **white-box** approach where the internals of the routing logic are implicitly visible to the author of the test.

Getting ready

The Java code for this recipe is located in the `org.camelcookbook.examples.testing.notifybuilder` package.

How to do it...

In order to do event-based testing in Camel, perform the following steps:

1. In a test method, instantiate a `NotifyBuilder` instance associated with the Camel context that you would like to test:

```
NotifyBuilder notify = new NotifyBuilder(camelContext)
  .from("activemq:in")
  .whenDone(1)
  .whenBodiesDone("Modified: testMessage")
  .create();
```

2. Send a message into the route using the endpoint technology. This is quite different to the approach that we have used so far where we used a Camel `ProducerTemplate` instance. Now we want to exercise the endpoint's consuming capabilities, whereas the `ProducerTemplate` approach skips that. In our test, we use a Spring `JmsTemplate` that is connected to same ActiveMQ broker used by the route:

```
jmsTemplate.send("in", new MessageCreator() {
    @Override
    public Message createMessage(Session session)
        throws JMSException {
      TextMessage textMessage =
          session.createTextMessage("testMessage");
      return textMessage;
    }
  });
```

3. Use a standard JUnit `assert` statement to determine whether the `NotifyBuilder` instance raises an event within a specified period of time:

   ```
   assertTrue(notify.matches(10, TimeUnit.SECONDS));
   ```

How it works...

The `NotifyBuilder` DSL `from()` statement locates a route by its starting endpoint. The `when..()` statements tell the builder to send an event when the conditions described have been matched. The `create()` method completes the construction of the builder, putting it into a state prepared to send events.

The builder's `matches()` method returns a `boolean` value that indicates whether the conditions as outlined were satisfied exchanges flowing through the route in the time specified. If the specified time expires without the conditions having been met, `matches()` will return `false`; otherwise `true` will be returned as soon as the conditions are met.

The `from(startingEndpoint)` syntax can be confusing to maintainers of your code, so a `fromRoute(id)` method is also provided that makes the intent of the builder somewhat clearer.

The `whenDone(number)` method requests that an event be raised when *at least* the specified number of exchanges have been successfully processed through the route. Other alternatives include `whenFailed(int)`, which expects at least the specified number of exceptions to have been thrown, and `whenComplete(int)`, which includes both succeeded and failed exchanges processed. All of these apply to result conditions once the message has fully been processed through a route. The `whenReceived(int)` method matches messages at the start of the route.

The `Done/Failed/Complete/Received` terminology is used in other condition methods, here checking whether an exact number of messages have been processed:

```
whenExactlyComplete(int)
whenExactlyDone(int)
whenExactlyFailed(int)
```

You can also check whether a particular message flowing through the route was successful. The index refers to the sequence of the message assigned as it is processed from the starting endpoint:

```
whenDoneByIndex(int)
```

There's more...

The `NotifyBuilder` can be set up to test more complex scenarios, such as checking whether certain messages reached an endpoint within a route:

```
new NotifyBuilder(camelContext)
  .from("activemq:in")
  .whenDone(1)
  .wereSentTo("activemq:out")
  .create();
```

Here, the `whenDone()` and `wereSentTo()` conditions are considered as being cumulative. You can use the Boolean `and()`, `or()`, and `not()` operators within the DSL to build up more complex checks. For example, here we check whether one message failed, and another succeeded:

```
new NotifyBuilder(camelContext)
  .from("activemq:in")
  .whenDone(1).wereSentTo("activemq:out")
  .and().whenFailed(1)
  .create();
```

Predicates, using Camel Expression Languages such as Simple and XPath, can also be used in your expressions:

```
whenAllDoneMatches(predicate)
whenAnyDoneMatches(predicate)
whenAllReceivedMatches(predicate)
whenAnyReceivedMatches(predicate)
```

You can use the `filter()` method to perform checks against certain messages only:

```
new NotifyBuilder(camelContext)
  .from("activemq:in")
  .whenExactlyDone(1).filter().simple("${body} contains 'test'")
  .create();
```

It is also possible to use the `NotifyBuilder` in conjunction with the `MockEndpoint` DSL to get a finer granularity of assertions over the message content:

```
MockEndpoint mock = camelContext.getEndpoint(
    "mock:nameDoesNotMatter", MockEndpoint.class);
mock.message(0).inMessage().contains(messageText);
mock.message(0).header("count").isNull();

NotifyBuilder notify = new NotifyBuilder(camelContext)
```

```
    .from("activemq:in")
    .whenDone(1).wereSentTo("activemq:out")
    .whenDoneSatisfied(mock)
    .create();
```

Here, the expectations set on the mock endpoint will be tested against the exchange state at the end of the route's processing.

Note that the name of mock endpoint interface does not correspond to an actual endpoint within the route. It is simply used to get a handle on the DSL for the `NotifyBuilder`.

See also

- NotifyBuilder: `http://camel.apache.org/notifybuilder.html`