Testing

Testing

Testing is a crucial activity in any piece of software development or integration. Typically Camel Riders use various different technologies wired together in a variety of patterns with different expression languages together with different forms of Bean Integration and Dependency Injection so its very easy for things to go wrong! . Testing is the crucial weapon to ensure that things work as you would expect.

Camel is a Java library so you can easily wire up tests in whatever unit testing framework you use (JUnit 3.x (deprecated), 4.x, or TestNG). However the Camel project has tried to make the testing of Camel as easy and powerful as possible so we have introduced the following features.

Testing Mechanisms

The following mechanisms are supported:

Name	Component	Description
Camel Test	`camel-test`	Is a standalone Java library letting you easily create Camel test cases using a single Java class for all your configuration and routing without using CDI, Spring or Guice for Dependency Injection which does not require an in-depth knowledge of Spring + Spring Test or Guice. Supports JUnit 3.x (deprecated) and JUnit 4.x based tests.
CDI Testing	`camel-test-cdi`	Provides a JUnit 4 runner that bootstraps a test environment using CDI so that you don't have to be familiar with any CDI testing frameworks and can concentrate on the testing logic of your Camel CDI applications. Testing frameworks like Arquillian or PAX Exam, can be used for more advanced test cases, where you need to configure your system under test in a very fine-grained way or target specific CDI containers.
Spring Testing	`camel-test-spring`	Supports JUnit 3.x (deprecated) or JUnit 4.x based tests that bootstrap a test environment using Spring without needing to be familiar with Spring Test. The plain JUnit 3.x/4.x based tests work very similar to the test support classes in **`camel-test`**. Also supports Spring Test based tests that use the declarative style of test configuration and injection common in Spring Test. The Spring Test based tests provide feature parity with the plain JUnit 3.x/4.x based testing approach. **Note**: **`camel-test-spring`** is a new component from **Camel 2.10**. For older Camel release use **`camel-test`** which has built-in Spring Testing.
Blueprint Testing	`camel-test-blueprint`	**Camel 2.10:** Provides the ability to do unit testing on blueprint configurations
Guice	`camel-guice`	**Deprecated** Uses Guice to dependency inject your test classes
Camel TestNG	`camel-testng`	**Deprecated** Supports plain TestNG based tests with or without CDI, Spring or Guice for Dependency Injection which does not require an in-depth knowledge of CDI, Spring + Spring Test or Guice. From **Camel 2.10**: this component supports Spring Test based tests that use the declarative style of test configuration and injection common in Spring Test and described in more detail under Spring Testing.

In all approaches the test classes look pretty much the same in that they all reuse the Camel binding and injection annotations.

Camel Test Example

Here is the Camel Test example:{snippet:lang=java|id=example|url=camel/trunk/components/camel-test/src/test/java/org/apache/camel/test/patterns/FilterTest.java}Notice how it derives from the Camel helper class **`CamelTestSupport`** but has no CDI, Spring or Guice dependency injection configuration but instead overrides the **`createRouteBuilder()`** method.

CDI Test Example

Here is the CDI Testing example:{snippet:lang=java|id=example|url=camel/trunk/components/camel-test-cdi/src/test/java/org/apache/camel/test/cdi/FilterTest.java}You can find more testing patterns illustrated in the **`camel-example-cdi-test`** example and the test classes that come with it.

Spring Test with XML Config Example

Here is the Spring Testing example using XML Config:{snippet:lang=java|id=example|url=camel/trunk/components/camel-spring/src/test/java/org/apache/camel/spring/patterns/FilterTest.java}Notice that we use **`@DirtiesContext`** on the test methods to force Spring Testing to automatically reload the `CamelContext` after each test method - this ensures that the tests don't clash with each other, e.g., one test method sending to an endpoint that is then reused in another test method.

Also note the use of **`@ContextConfiguration`** to indicate that by default we should look for the `FilterTest-context.xml` on the classpath to configure the test case which looks like this:{snippet:lang=xml|id=example|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/patterns/FilterTest-context.xml}

Spring Test with Java Config Example

Here is the Spring Testing example using Java Config.

For more information see Spring Java Config.{snippet:lang=java|id=example|url=camel/trunk/components/camel-spring-javaconfig/src/test/java/org/apache/camel/spring/javaconfig/patterns/FilterTest.java}This is similar to the XML Config example above except that there is no XML file and instead the nested **ContextConfig** class does all of the configuration; so your entire test case is contained in a single Java class. We currently have to reference by class name this class in the **@ContextConfiguration** which is a bit ugly. Please vote for SJC-238 to address this and make Spring Test work more cleanly with Spring JavaConfig.

Its totally optional but for the **ContextConfig** implementation we derive from **SingleRouteCamelConfiguration** which is a helper Spring Java Config class which will configure the **CamelContext** for us and then register the **RouteBuilder** we create.

Since **Camel 2.11.0** you can use the **CamelSpringJUnit4ClassRunner** with **CamelSpringDelegatingTestContextLoader** like example using Java Config with `CamelSpringJUnit4ClassRunner`:{snippet:lang=java|id=example|url=camel/trunk/components/camel-spring-javaconfig/src/test/java/org/apache/camel/spring/javaconfig/test/CamelSpringDelegatingTestContextLoaderTest.java}

Spring Test with XML Config and Declarative Configuration Example

Here is a Camel test support enhanced Spring Testing example using XML Config and pure Spring Test based configuration of the Camel Context:{snippet:lang=java|id=e1|url=camel/trunk/components/camel-test-spring/src/test/java/org/apache/camel/test/spring/CamelSpringJUnit4ClassRunnerPlainTest.java}Notice how a custom test runner is used with the **@RunWith** annotation to support the features of **CamelTestSupport** through annotations on the test class. See Spring Testing for a list of annotations you can use in your tests.

Blueprint Test

Here is the Blueprint Testing example using XML Config:{snippet:lang=java|id=example|url=camel/trunk/components/camel-test-blueprint/src/test/java/org/apache/camel/test/blueprint/DebugBlueprintTest.java}Also notice the use of **getBlueprintDescriptors** to indicate that by default we should look for the `camelContext.xml` in the package to configure the test case which looks like this:{snippet:lang=xml|id=example|url=camel/trunk/components/camel-test-blueprint/src/test/resources/org/apache/camel/test/blueprint/camelContext.xml}

Testing Endpoints

Camel provides a number of endpoints which can make testing easier.

Name	Description
DataSet	For load & soak testing this endpoint provides a way to create huge numbers of messages for sending to Components and asserting that they are consumed correctly
Mock	For testing routes and mediation rules using mocks and allowing assertions to be added to an endpoint
Test	Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint

The main endpoint is the Mock endpoint which allows expectations to be added to different endpoints; you can then run your tests and assert that your expectations are met at the end.

Stubbing out physical transport technologies

If you wish to test out a route but want to avoid actually using a real physical transport (for example to unit test a transformation route rather than performing a full integration test) then the following endpoints can be useful.

Name	Description
Direct	Direct invocation of the consumer from the producer so that single threaded (non-SEDA) in VM invocation is performed which can be useful to mock out physical transports
SEDA	Delivers messages asynchronously to consumers via a java.util.concurrent.BlockingQueue which is good for testing asynchronous transports
Stub	Works like SEDA but does not validate the endpoint URI, which makes stubbing much easier.

Testing existing routes

Camel provides some features to aid during testing of existing routes where you cannot or will not use Mock etc. For example you may have a production ready route which you want to test with some 3rd party API which sends messages into this route.

Name	Description
NotifyBuilder	Allows you to be notified when a certain condition has occurred. For example when the route has completed five messages. You can build complex expressions to match your criteria when to be notified.

Advice With	Allows you to **advice** or **enhance** an existing route using a RouteBuilder style. For example you can add interceptors to intercept sending outgoing messages to assert those messages are as expected.