Spring Testing

Spring Testing

Testing is a crucial part of any development or integration work. The Spring Framework offers a number of features that makes it easy to test while using Spring for Inversion of Control which works with JUnit 3.x, JUnit 4.x, and TestNG.

We can use Spring for IoC and the Camel Mock and Test endpoints to create sophisticated integration/unit tests that are easy to run and debug inside your IDE. There are three supported approaches for testing with Spring in Camel.

Name	Testing Frameworks Supported	Description	Required Camel Test Dependencies
`CamelSpringTestSupport`	• JUnit 3.x (deprecated) • JUnit 4.x • TestNG - **Camel 2.8**	Provided by: • **`org.apache.camel.test.CamelSpringTestSupport`** • **`org.apache.camel.test.junit4.CamelSpringTestSupport`** • **`org.apache.camel.testng.CamelSpringTestSupport`** These base classes provide feature parity with the simple **`CamelTestSupport`** classes from Camel Test but do not support Spring annotations on the test class such as **`@Autowired`**, **`@DirtiesContext`**, and **`@ContextConfiguration`**.	• JUnit 3.x (deprecated) - `camel-test-spring` • JUnit 4.x - `camel-test-spring` • TestNG - `camel-test-ng`
Plain Spring Test	• JUnit 3.x • JUnit 4.x • TestNG	Either extend the abstract base classes: • **`org.springframework.test.context.junit38.AbstractJUnit38SpringContextTests`** • **`org.springframework.test.context.junit38.AbstractJUnit4SpringContextTests`** • etc. provided in Spring Test or use the Spring Test JUnit4 runner. These approaches support both the Camel annotations and Spring annotations. However, they do NOT have feature parity with: • **`org.apache.camel.test.CamelTestSupport`** • **`org.apache.camel.test.junit4.CamelTestSupport`** • **`org.apache.camel.testng.CamelSpringTestSupport`**	• JUnit 3.x (deprecated) - None • JUnit 4.x - None • TestNG - None
Camel Enhanced Spring Test	• JUnit 4.x - **Camel 2.10** • TestNG - **Camel 2.10**	Either: • use the **`org.apache.camel.test.junit4.CamelSpringJUnit4ClassRunner`** runner with the **`@RunWith`** annotation, • or extend **`org.apache.camel.testng.AbstractCamelTestNGSpringContextTests`** to enable feature parity with **`org.apache.camel.test.CamelTestSupport`** and **`org.apache.camel.test.junit4.CamelTestSupport`**. These classes support the full suite of Spring Test annotations such as **`@Autowired`**, **`@DirtiesContext`**, and **`@ContextConfiguration`**.	JUnit 3.x (deprecated) - `camel-test-spring` JUnit 4.x - `camel-test-spring` TestNG - `camel-test-ng`

CamelSpringTestSupport

The following Spring test support classes:

- **`org.apache.camel.test.CamelSpringTestSupport`**
- **`org.apache.camel.test.junit4.CamelSpringTestSupport`**, and
- **`org.apache.camel.testng.CamelSpringTestSupport`**

extend their non-Spring aware counterparts:

- **`org.apache.camel.test.CamelTestSupport`**
- **`org.apache.camel.test.junit4.CamelTestSupport`**, and
- **`org.apache.camel.testng.CamelTestSupport`**

and deliver integration with Spring into your test classes.

Instead of instantiating the **`CamelContext`** and routes programmatically, these classes rely on a Spring context to wire the needed components together. If your test extends one of these classes, you must provide the Spring context by implementing the following method.

javaprotected abstract AbstractApplicationContext createApplicationContext();

You are responsible for the instantiation of the Spring context in the method implementation. All of the features available in the non-Spring aware counterparts from Camel Test are available in your test.

Plain Spring Test

In this approach, your test classes directly inherit from the Spring Test abstract test classes or use the JUnit 4.x test runner provided in Spring Test. This approach supports dependency injection into your test class and the full suite of Spring Test annotations. However, it does not support the features provided by the **`CamelSpringTestSupport`** classes.

Plain Spring Test using JUnit 3.x with XML Config Example

Here is a simple unit test using JUnit 3.x support from Spring Test 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 notice the use of **`@ContextConfiguration`** to indicate that by default we should look for the file FilterTest-context.xml on the classpath to configure the test case. The test context looks like:{snippet:lang=xml|id=example|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/spring/patterns/FilterTest-context.xml}This test will load a Spring XML configuration file called **`FilterTest-context.xml`** from the classpath in the same package structure as the **`FilterTest`** class and initialize it along with any Camel routes we define inside it, then inject the **`CamelContext`** instance into our test case.

For instance, like this maven folder layout:

src/test/java/org/apache/camel/spring/patterns/FilterTest.java src/test/resources/org/apache/camel/spring/patterns/FilterTest-context.xml

Plain Spring Test Using JUnit 4.x With Java Config Example

You can completely avoid using an XML configuration file by using Spring Java Config. Here is a unit test using JUnit 4.x support from Spring Test using 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.

Plain Spring Test Using JUnit 4.0.x Runner With XML Config

You can avoid extending Spring classes by using the **`SpringJUnit4ClassRunner`** provided by Spring Test. This custom JUnit runner means you are free to choose your own class hierarchy while retaining all the capabilities of Spring Test.

This is for Spring 4.0.x. If you use Spring 4.1 or newer, then see the next section.
java@RunWith(SpringJUnit4ClassRunner.class) @ContextConfiguration public class MyCamelTest {   @Autowired   protected CamelContext camelContext;   @EndpointInject(uri = "mock:foo")   protected MockEndpoint foo; @Test @DirtiesContext   public void testMocksAreValid() throws Exception { // ...   foo.message(0).header("bar").isEqualTo("ABC");   MockEndpoint.assertIsSatisfied(camelContext);   } }

Plain Spring Test Using JUnit 4.1.x Runner With XML Config

You can avoid extending Spring classes by using the **`SpringJUnit4ClassRunner`** provided by Spring Test. This custom JUnit runner means you are free to choose your own class hierarchy while retaining all the capabilities of Spring Test.

From **Spring 4.1**, you need to use the **`@BootstrapWith`** annotation to configure it to use Camel testing, as shown below.
java@RunWith(CamelSpringJUnit4ClassRunner.class) @BootstrapWith(CamelTestContextBootstrapper.class) @ContextConfiguration public class MyCamelTest {   @Autowired   protected CamelContext camelContext;   @EndpointInject(uri = "mock:foo")   protected MockEndpoint foo; @Test @DirtiesContext   public void testMocksAreValid() throws Exception { // ...   foo.message(0).header("bar").isEqualTo("ABC");   MockEndpoint.assertIsSatisfied(camelContext);   } }

Camel Enhanced Spring Test

Using the **`org.apache.camel.test.junit4.CamelSpringJUnit4ClassRunner`** runner with the **`@RunWith`** annotation or extending **`org.apache.camel.testng.AbstractCamelTestNGSpringContextTests`** provides the full feature set of Spring Test with support for the feature set provided in the **`CamelTestSupport`** classes.

A number of Camel specific annotations have been developed in order to provide for declarative manipulation of the Camel context(s) involved in the test. These annotations free your test classes from having to inherit from the **`CamelSpringTestSupport`** classes and also reduce the amount of code required to customize the tests.

Annotation Class	Applies To	Description	Default Behavioir If Not Present	Default Behavior If Present
`org.apache.camel.test.spring.DisableJmx`	`Class`	Indicates if JMX should be globally disabled in the CamelContexts that are bootstrapped during the test through the use of Spring Test loaded application contexts.	JMX is disabled	JMX is disabled
`org.apache.camel.test.spring.ExcludeRoutes`	`Class`	Indicates if certain route builder classes should be excluded from discovery. Initializes a **`org.apache.camel.spi.PackageScanClassResolver`** to exclude a set of given classes from being resolved. Typically this is used at test time to exclude certain routes, which might otherwise be just noisy, from being discovered and initialized.	Not enabled and no routes are excluded	No routes are excluded

`org.apache.camel.test.spring.LazyLoadTypeConverters`	`Class`	**Deprecated.** Indicates if the CamelContexts that are bootstrapped during the test through the use of Spring Test loaded application contexts should use lazy loading of type converters.	Type converters are not lazy loaded	Type converters are not lazy loaded
`org.apache.camel.test.spring.MockEndpoints`	`Class`	Triggers the auto-mocking of endpoints whose URIs match the provided filter. The default filter is **`"*"`** which matches all endpoints. See **`org.apache.camel.impl.InterceptSendToMockEndpointStrategy`** for more details on the registration of the mock endpoints.	Not enabled	All endpoints are sniffed and recorded in a mock endpoint.
`org.apache.camel.test.spring.MockEndpointsAndSkip`	`Class`	Triggers the auto-mocking of endpoints whose URIs match the provided filter. The default filter is **`"*"`**, which matches all endpoints. See org.apache.camel.impl.InterceptSendToMockEndpointStrategy for more details on the registration of the mock endpoints. This annotation will also skip sending the message to matched endpoints as well.	Not enabled	All endpoints are sniffed and recorded in a mock endpoint. The original endpoint is not invoked.
`org.apache.camel.test.spring.ProvidesBreakpoint`	`Method`	Indicates that the annotated method returns an **`org.apache.camel.spi.Breakpoint`** for use in the test. Useful for intercepting traffic to all endpoints or simply for setting a break point in an IDE for debugging. The method must be public, static, take no arguments, and return **`org.apache.camel.spi.Breakpoint`**.	N/A	The returned **`Breakpoint`** is registered in the CamelContext(s)
`org.apache.camel.test.spring.ShutdownTimeout`	`Class`	Indicates to set the shutdown timeout of all CamelContexts instantiated through the use of Spring Test loaded application contexts. If no annotation is used, the timeout is automatically reduced to 10 seconds by the test framework.	10 seconds	10 seconds
`org.apache.camel.test.spring.UseAdviceWith`	`Class`	Indicates the use of **`adviceWith()`** within the test class. If a class is annotated with this annotation and **`UseAdviceWith#value()`** returns true, any CamelContexts bootstrapped during the test through the use of Spring Test loaded application contexts will not be started automatically. The test author is responsible for injecting the Camel contexts into the test and executing **`CamelContext#start()`** on them at the appropriate time after any advice has been applied to the routes in the CamelContext(s).	CamelContexts do not automatically start.	CamelContexts do not automatically start.
`org.apache.camel.test.spring.UseOverridePropertiesWithPropertiesComponent`	`Method`	**Camel 2.16:**Indicates that the annotated method returns a **`java.util.Properties`** for use in the test, and that those properties override any existing properties configured on the **`PropertiesComponent`**.		Override properties

The following example illustrates the use of the **`@MockEndpoints`** annotation in order to setup mock endpoints as interceptors on all endpoints using the Camel Log component and the **`@DisableJmx`** annotation to enable JMX which is disabled during tests by default.

Note: we still use the **`@DirtiesContext`** annotation to ensure that the CamelContext, routes, and mock endpoints are reinitialized between test methods.java@RunWith(CamelSpringJUnit4ClassRunner.class) @BootstrapWith(CamelTestContextBootstrapper.class) @ContextConfiguration @DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD) @MockEndpoints("log:*") @DisableJmx(false) public class CamelSpringJUnit4ClassRunnerPlainTest { @Autowired protected CamelContext camelContext2; protected MockEndpoint mockB; @EndpointInject(uri = "mock:c", context = "camelContext2") protected MockEndpoint mockC; @Produce(uri = "direct:start2", context = "camelContext2") protected ProducerTemplate start2; @EndpointInject(uri = "mock:log:org.apache.camel.test.junit4.spring", context = "camelContext2") protected MockEndpoint mockLog; @Test public void testPositive() throws Exception { mockC.expectedBodiesReceived("David"); mockLog.expectedBodiesReceived("Hello David"); start2.sendBody("David"); MockEndpoint.assertIsSatisfied(camelContext); }

Adding More Mock Expectations

If you wish to programmatically add any new assertions to your test you can easily do so with the following. Notice how we use **`@EndpointInject`** to inject a Camel endpoint into our code then the Mock API to add an expectation on a specific message.

java@ContextConfiguration public class MyCamelTest extends AbstractJUnit38SpringContextTests { @Autowired protected CamelContext camelContext; @EndpointInject(uri = "mock:foo") protected MockEndpoint foo; public void testMocksAreValid() throws Exception { // lets add more expectations foo.message(0).header("bar").isEqualTo("ABC"); MockEndpoint.assertIsSatisfied(camelContext); } }

Further Processing the Received Messages

Sometimes once a Mock endpoint has received some messages you want to then process them further to add further assertions that your test case worked as you expect.

So you can then process the received message exchanges if you like...

java@ContextConfiguration public class MyCamelTest extends AbstractJUnit38SpringContextTests { @Autowired protected CamelContext camelContext; @EndpointInject(uri = "mock:foo") protected MockEndpoint foo; public void testMocksAreValid() throws Exception { // lets add more expectations... MockEndpoint.assertIsSatisfied(camelContext); // now lets do some further assertions List<Exchange> list = foo.getReceivedExchanges(); for (Exchange exchange : list) { Message in = exchange.getIn(); // ... } } }

Sending and Receiving Messages

It might be that the Enterprise Integration Patterns you have defined in either Spring XML or using the Java DSL do all of the sending and receiving and you might just work with the Mock endpoints as described above. However sometimes in a test case its useful to explicitly send or receive messages directly.

To send or receive messages you should use the Bean Integration mechanism. For example to send messages inject a **`ProducerTemplate`** using the **`@EndpointInject`** annotation then call the various send methods on this object to send a message to an endpoint. To consume messages use the **`@MessageDriven`** annotation on a method to have the method invoked when a message is received.

```
javapublic class Foo { @EndpointInject(uri = "activemq:foo.bar") ProducerTemplate producer; public void doSomething() { // lets send a message! producer.
sendBody("<hello>world!</hello>"); } // lets consume messages from the 'cheese' queue @MessageDriven(uri="activemq:cheese") public void onCheese
(String name) { // ... } }
```

See Also

- A real example test case using Mock and Spring along with its Spring XML
- Bean Integration
- Mock endpoint
- Test endpoint