9

Testing

This chapter covers

- Introducing and using the Camel Test Kit
- Unit-testing Camel
- Testing with mocks
- Simulating components and errors
- Amending routes before testing
- Integration and system testing
- Using third-party test frameworks with Camel

Integrated systems are notoriously difficult to test. Systems being built and integrated today often involve a myriad of systems ranging from legacy systems, closed source commercial products, homegrown applications, and so on. These systems often have no built-in support for automated testing.

The poor folks who are tasked with testing such integrated systems often have no other choice than to play through manual use cases—triggering events from one window/terminal and then crossing their fingers and watching the results in the affected systems. And by *watching*, we mean looking at log files, checking databases, or any other manual procedure to verify the results. When they move on to the next use case, they have to *reset the system* to ensure its correct state before carrying on with the testing.

During development, the systems being integrated are often test systems that are from other teams or are available only in production. If the systems aren't network connected, internal company procedures may hold back a speedy resolution. With the higher number of systems involved, it becomes more difficult for developers to build and test their work.

Does this sound familiar? It should, because these problems have been around for decades, and people have found ways around them in order to do their jobs. One such way is to stub out other systems and not rely on their physical implementations. Camel has answers for this with a built-in test kit that allows you to treat integration points as components that can be switched out with local testable implementations.

This chapters starts by introducing the Camel Test Kit and teaching you to perform Camel testing with various runtime platforms such as stand-alone Java, Spring Boot, CDI, OSGi, and Java EE servers such as WildFly.

At this point in this book, you've likely looked at the source code examples and seen that the vast majority of the code is driven by unit tests. Therefore, you may have seen that some of these tests are using the Camel mock component. We'll dive deeper into what the mock component is, what functionality it offers for testing, and how to get more out of it in your tests.

We've said it before: integration is hard. Building and integrating systems would be easier if nothing ever went wrong. But what about when things do go wrong? How do you test your systems when a remote system is unavailable, when invalid data is returned, or when a database starts failing with SQL violation errors? This chapter covers how to simulate such error conditions in your tests.

Integration systems start to add value to your business only when they become live in production. You may wonder how you can build Camel routes that run in production but are testable both locally and in test environments. In this chapter, you'll learn the strategies that the Camel Test Kit offers for testing Camel routes that are built as production-ready routes.

At the end of the chapter, we'll look at three third-party testing frameworks for doing integration testing and learn how to use them with your Camel applications.

Other testing disciplines that we don't cover in this book are worth keeping in mind, such as load, performance, and stress testing. Recently, companies have started seeing the benefits of having a continuous integration

and deployment platform that helps drive test automation and makes organizations more agile while reducing time to production.

Testing starts with unit tests. And a good way to perform unit testing on a Camel application is to start the application, send messages to the application, and verify that the messages are routed as expected. This is illustrated in figure 9.1. You send a message to the application, which transforms the message to another format and returns the output. You can then verify that the output is as expected.

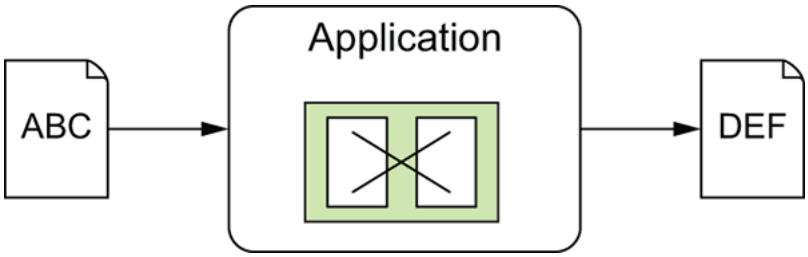

Figure 9.1 Testing a Camel application by sending a message to the application and then verifying the returned output

This is how the Camel Test Kit is used for testing. You'll learn to set up expectations as preconditions for your unit tests, start the tests by sending in messages, and verify the results to determine whether the tests passed.

9.1 Introducing the Camel Test Kit

Camel provides rich facilities for testing your projects, using the Camel Test Kit. This kit is used heavily for testing Camel itself (Camel is eating its own dog food). Figure 9.2 gives a high-level overview of the Camel Test Kit.

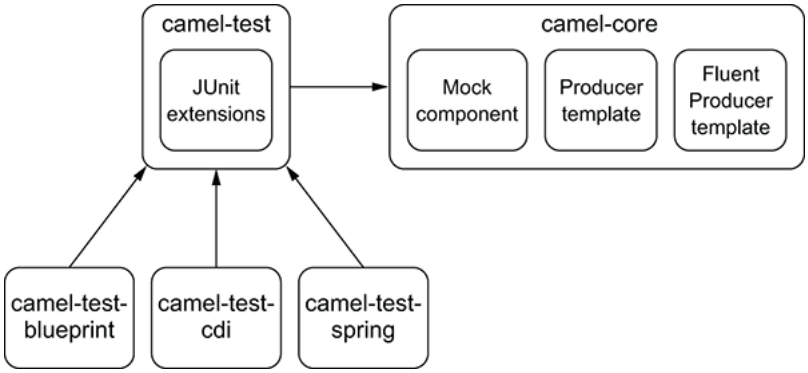

Figure 9.2 The Camel Test Kit is comprised of the camel-test JAR for testing with Java in general, and three specialized JARs for testing with with OSGi Blueprint, CDI, and Spring. Inside camel-core you'll find the mock component, `ProducerTemplate`, and `FluentProducerTemplate`, which are frequently used with unit testing.

The test kit has four main parts. The camel-test JAR is the core test module that includes the JUnit extensions as a number of classes on top of

JUnit that make unit testing with Camel much easier. We cover them in the next section. The camel-test JAR can be used for testing standalone Camel applications. Testing Camel in other environments requires specialized camel-test modules. For OSGi Blueprint testing, you should use camel-test-blueprint, for CDI testing use camel-test-cdi, and for Camel with Spring use camel-test-spring. This section focuses on testing standalone Camel applications using plain Java. Section 9.2 covers testing Camel with Spring, OSGi Blueprint, and CDI.

The Camel Test Kit uses the mock component from camel-core, covered in section 9.3. And you're already familiar with `ProducerTemplate` and `FluentProducerTemplate` that makes it easy to send messages to your Camel routes.

Let's now look at the Camel JUnit extensions and see how to use them to write Camel unit tests.

9.1.1 Using the Camel JUnit extensions

What are the Camel JUnit extensions? They are nine classes in a small JAR file, camel-test.jar, that ships with Camel. Out of those nine classes are three that are intended to be used by end users. They are listed in table 9.1.

Table 9.1 End-user-related classes in the Camel Test Kit, provided in camel-test.jar

Class	Description
`org.apache.camel.test.junit4.TestSupport`	Abstract base class with additional builder and assertion methods.
`org.apache.camel.test.junit4.CamelTestSupport`	Base test class prepared for testing Camel routes. This is the test class you should use when testing your Camel routes.
`org.apache.camel.test.AvailablePortFinder`	Helper class to find unused TCP port numbers that your unit tests can use when testing with TCP connections.

To get started with the Camel Test Kit, you'll use the following route for testing:

```
from("file:inbox").to("file:outbox");
```

This is the “Hello World” example for integration kits that moves files from one folder to another. How do you go about unit testing this route?

You could do it the traditional way and write unit test code with the plain JUnit API. This would require at least 30 lines of code, because the API for file handling in Java is low level, and you need a fair amount of code when working with files.

An easier solution is to use the Camel Test Kit. In the next couple of sections, you’ll work with the `CamelTestSupport` class—the easiest to start with.

9.1.2 Using camel-test to test Java Camel routes

In this chapter, we’ve kept the dependencies low when using the Camel Test Kit. All you need to include is the following dependency in the Maven pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test</artifactId>
  <version>2.20.1</version>
  <scope>test</scope>
</dependency>
```

Let’s try it. You want to build a unit test to test a Camel route that copies files from one directory to another. The unit test is shown in the following listing.

Listing 9.1 A first unit test using the Camel Test Kit

```
public class FirstTest extends CamelTestSupport {

  @Override
  protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
  public void configure() throws Exception {  1
```

❶

Defines route to test

```
        from("file://target/inbox")
          .to("file://target/outbox");
      }
    };
  }

  @Test
  public void testMoveFile() throws Exception {
    template.sendBodyAndHeader("file://target/inbox", "Hello World",
        Exchange.FILE_NAME, "hello.txt");  ❷
```

❷

Creates hello.txt file

```
    Thread.sleep(2000);

    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());  ❸
```

❸

Verifies file is moved

```
  }
}
```

The `FirstTest` class must extend the `org.apache.camel.junit4.CamelTestSupport` class to conveniently use the Camel Test Kit. By overriding the `createRouteBuilder` method, you can provide any route builder you wish. You use an inlined route builder, which allows you to write the route directly within the unit-test class. All you need to do is override the `configure` method ❶ and include your route.

The test methods are regular JUnit methods, so the method must be annotated with `@Test` to be included when testing. You'll notice that the code in this method is fairly short. Instead of using the low-level Java File API, this example uses Camel as a client by using `ProducerTemplate` to send a message to a file endpoint ❷, which writes the message as a file.

In the test, you sleep 2 seconds after dropping the file in the inbox folder; this gives Camel a bit of time to react and route the file. By default, Camel scans twice per second for incoming files, so you wait 2 seconds to be on the safe side. Finally, you assert that the file was moved to the outbox folder ❸.

The book's source code includes this example. You can try it on your own by running the following Maven goal from the chapter9/java directory:

```
mvn test -Dtest=FirstTest
```

When you run this example, it should output the result of the test as shown here:

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

This indicates that the test completed successfully; there are no failures or errors.

Improving the unit test

The unit test in listing 9.1 could be improved in a few areas, such as ensuring that the starting directory is empty and that the written file's content is what you expect.

The former is easy, because the `CamelTestSupport` class has a method to delete a directory. You can do this in the `setUp` method:

```
public void setUp() throws Exception {
    deleteDirectory("target/inbox");
    deleteDirectory("target/outbox");
    super.setUp();
}
```

Camel can also test the written file's content to ensure that it's what you expect. You may remember that Camel provides an elaborate type-converter system, and that this system goes beyond converting between simple types and literals. The Camel type system includes file-based converters, so there's no need to fiddle with the various cumbersome Java I/O file streams. All you need to do is ask the type-converter system to grab the file and return it to you as a `String`.

Just as you had access to the template in listing 9.1, the Camel Test Kit also gives you direct access to `CamelContext`. The `testMoveFile` method in listing 9.1 could have been written as follows:

```
@Test
public void testMoveFile() throws Exception {
  template.sendBodyAndHeader("file://target/inbox", "Hello World",
              Exchange.FILE_NAME, "hello.txt");

  Thread.sleep(2000);

  File target = new File("target/outbox/hello.txt");
  assertTrue("File not moved", target.exists());
```

Asserts the file is moved

```
  String content = context.getTypeConverter()
                          .convertTo(String.class, target);
  assertEquals("Hello World", content);
```

Asserts the file content is Hello World

```
}
```

Is there one thing in the test code that annoys you? Yeah, `Thread.sleep(2000)`, to wait for Camel to pick up and process the file. Camel can do better. You can use something called `NotifyBuilder` to set

up a precondition using the builder and then let it wait until the condition is satisfied before continuing. We cover `NotifyBuilder` in section 9.5.2, so we'll just quickly show you the revised code:

```
@Test
public void testMoveFile() throws Exception {
  NotifyBuilder notify = new NotifyBuilder(context)
                .whenDone(1).create();
```

Sets up precondition on NotifyBuilder

```
  template.sendBodyAndHeader("file://target/inbox", "Hello World",
              Exchange.FILE_NAME, "hello.txt");

  assertTrue(notify.matchesMockWaitTime());
```

NotifyBuilder is waiting for the precondition to be satisfied.

```
  File target = new File("target/outbox/hello.txt");
  assertTrue("File not moved", target.exists());
```

Asserts the file is moved

```
  String content = context.getTypeConverter()
                          .convertTo(String.class, target);
  assertEquals("Hello World", content);
```

Asserts the file content is Hello World

```
}
```

The book's source code includes this revised example. You can try it by running the following Maven goal from the chapter9/java directory:

```
mvn test -Dtest=FirstNoSleepTest
```

The preceding examples cover the case in which the route is defined in the unit-test class as an anonymous inner class. But what if you have a route defined in another class? How do you go about unit testing that route instead? Let's look at that next.

9.1.3 Unit testing an existing RouteBuilder class

Defining Camel routes in separate `RouteBuilder` classes is common, as in the `FileMoveRoute` class here:

```
public class FileMoveRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {
    from("file://target/inbox")
      .to("file://target/outbox");
  }
}
```

How could you unit test this route from the `FileMoveRoute` class? You don't want to copy the code from the `configure` method into a JUnit class. Fortunately, it's easy to set up unit tests that use the `FileMoveRoute`, as you can see here:

```
protected RouteBuilder createRouteBuilder() throws Exception {
    return new FileMoveRoute();
}
```

Yes, it's that simple! Just return a new instance of your route class.

This book's source code contains this example in the chapter9/java directory. You can try the example by using the following Maven goal:

```
mvn test -Dtest=ExistingRouteBuilderTest
```

Notice that the `FileMoveRoute` class is located in the src/main/java/camelinaction directory and isn't located in the test directory.

Now you've learned how to use `CamelTestSupport` for unit testing routes based on the Java DSL. You use `CamelTestSupport` from camel-test for testing standard Java-based Camel applications. But Camel can run in many platforms. Let's dive into covering the most popular choices.

9.2 Testing Camel with Spring, OSGi, and CDI

Camel applications are being used in many environments and together with various frameworks. Over the years, people have often run Camel with either Spring or OSGi Blueprint, and more recently in using CDI with Java EE applications. This section demonstrates how to start building unit tests with the following frameworks and platforms:

- Camel using Spring XML
- Camel using Spring Java Config
- Camel using Spring Boot
- Camel using OSGi Blueprint XML running in Apache Karaf/ServiceMix, or JBoss Fuse
- Camel using CDI running standalone using JBoss Weld CDI container
- Camel using CDI running in WildFly Swarm
- Camel using CDI running in WildFly

It's worth emphasizing that the goal of this section is to show you how to get a good start in testing Camel on these seven platforms. The focus is therefore on using a simple test case as an example to highlight how the testing is specific to the chosen platform. We'll dive deeper into testing topics such as integration and systems tests, and more elaborate test functionality from Camel, in the sections to follow.

9.2.1 Camel testing with Spring XML

You use camel-test-spring to test Spring-based Camel applications. In the Maven pom.xml file, add the following dependency:

```
<dependency>
  <groupId>org.apache.camel</groupId>
```

```
    <artifactId>camel-test-spring</artifactId>
    <version>2.20.1</version>
    <scope>test</scope>
  </dependency>
```

This section looks at unit-testing the route in the following listing.

Listing 9.2 A Spring-based version of the route in listing 9.1 (firststep.xml)

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://camel.apache.org/schema/spring
         http://camel.apache.org/schema/spring/camel-spring.xsd">

  <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="file://target/inbox"/>
      <to uri="file://target/outbox"/>
    </route>
  </camelContext>

</beans>
```

How can you unit-test this route? Ideally, you should be able to use unit tests regardless of the language used to define the route—whether using Java DSL or XML DSL, for example.

The camel-test-spring module is an extension to camel-test, which means all the test principles you learned in section 9.1 also apply. Camel is able to handle this; the difference between using `SpringCamelTestSupport` and `CamelTestSupport` is just a matter of how the route is loaded. The unit test in the following listing illustrates this point.

Listing 9.3 A first unit test using Spring XML routes

```
public class SpringFirstTest extends CamelSpringTestSupport {

  protected AbstractXmlApplicationContext createApplicationContext() {
```

```
    return new ClassPathXmlApplicationContext(
            "camelinaction/firststep.xml");  ❶
```

❶ Loads Spring XML file

```
  }

  @Test
  public void testMoveFile() throws Exception {
    template.sendBodyAndHeader("file://target/inbox",
                "Hello World", Exchange.FILE_NAME, "hello.txt");

    Thread.sleep(2000);

    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());
    String content = context.getTypeConverter()
                            .convertTo(String.class, target);
    assertEquals("Hello World", content);
  }
}
```

You extend the `CamelSpringTestSupport` class so you can unit-test with Spring XML–based routes. You need to use a Spring-based mechanism to load the routes ❶; you use the `ClassPathXmlApplicationContext`, which loads your route from the classpath. This mechanism is entirely Spring based, so you can also use the `FileSystemXmlApplicationContext`, include multiple XML files, and so on; Camel doesn't impose any restrictions. The `testMoveFile` method is exactly the same as it was when testing using Java DSL in section 9.1, which means you can use the same unit-testing code regardless of how the route is defined.

This book's source code contains this example in the chapter9/spring-xml directory; you can try the example by using the following Maven goal:

```
mvn test -Dtest=SpringFirstTest
```

Unit testing with @RunWith

The test class in listing 9.3 extends the `CamelSpringTestSupport` class. But you can also write Camel unit tests without extending a base class, which is a more modern style with JUnit. The following listing shows how to write the unit test in a more modern style.

Listing 9.4 Using a modern JUnit style to write a Camel Spring unit test

```
@RunWith(CamelSpringRunner.class)    (1)
```

(1) Runs the test using Camel Spring runner

```
@ContextConfiguration(locations = {"firststep.xml"})    (2)
```

(2) Declares the location of the Spring XML file relative to this unit test

```
public class SpringFirstRunWithTest {

  @Autowired
  private CamelContext context;    (3)
```

(3) Dependency injects CamelContext

```
  @Autowired
  private ProducerTemplate template;    (4)
```

(4) Dependency injects ProducerTemplate

```

  @Test
```

```
  public void testMoveFile() throws Exception {    ❺
```

❺

Test method same as in listing 9.3

```
    template.sendBodyAndHeader("file://target/inbox",
                "Hello World", Exchange.FILE_NAME, "hello.txt");

    Thread.sleep(2000);

    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());
    String content = context.getTypeConverter()
                            .convertTo(String.class, target);
    assertEquals("Hello World", content);
  }
}
```

Instead of extending a base class, the test class is annotated with `@RunWith(CamelSpringRunner.class)` ❶. The `CamelSpringRunner` is an extension to Spring's `SpringJUnit4ClassRunner` that integrates Camel with Spring and allows you to use additional Camel annotations in the test. We cover such use cases later in this chapter.

The `@ContextConfiguration` annotation is used to configure where the Spring XML file is located in the classpath ❷. Notice that the location is relative to the current class, so the firststep.xml file is also located in the `camelinaction` package.

To use Camel during the unit test, you need to dependency inject both `CamelContext` ❸ and `ProducerTemplate` ❹. The test method is un-changed ❺.

This example is also provided with the source code in the chapter9/spring-xml directory, and you can try the example by running the following Maven goal:

```
mvn test -Dtest=SpringFirstRunWithTest
```

Now let's move on to testing Spring applications that are using Java Config instead of XML.

9.2.2 Camel testing with Spring Java Config

The Spring Framework started out mainly using XML configuration files, which became a popular choice of building applications. Camel works well with XML by offering the XML DSL so you can define Camel routes directly in your Spring XML files. But recently, an alternative configuration using plain Java has become popular as well—with a lot of thanks to the rising popularity of Spring Boot.

When using Camel with Spring Java Config, you must remember to add the following dependency to your Maven pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-javaconfig</artifactId>
  <version>2.20.1</version>
</dependency>
```

You can use Spring Java Config both standalone and with Spring. We cover both scenarios in this and the following section.

The example we use is `FileMoveRoute`, as shown here:

```
@Component    (1)
```

---

(1) Annotates the class with @Component to make Spring discover the class easily

---

```
public class FileMoveRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {
    from("file://target/inbox")
      .to("file://target/outbox");
  }
}
```

The `FileMoveRoute` class is a plain Camel route that has been annotated with `@Component` ❶. This enables Spring Java Config to discover the route during startup and automatically create an instance that in turn Camel discovers from Spring and automatically includes the route.

To bootstrap and run Camel with Spring Java Config, you need the following:

- A Java main class to start the application
- An `@Configuration` class to bootstrap Spring Java Config
- You can combine both in the same class, as shown in the following listing.

Listing 9.5 Java main class to bootstrap Spring Java Config with Camel

```
@Configuration    ❶
```

❶ Marks this class as a Spring Java configuration class

```
@ComponentScan("camelinaction")    ❷
```

❷ Tells Spring to scan in the package for classes with Spring annotations

```
public class MyApplication extends CamelConfiguration {    ❸
```

❸ To automatically enable Camel

```
  public static void main(String[] args) throws Exception {
    Main main = new Main();    ❹
```

❹

Main class to easily start the application and keep the JVM running

```
    main.setConfigClass(MyApplication.class);    ❺
```

❺

Uses this class as the configuration class

```
    main.run();
  }
}
```

The `MyApplication` class extends `CamelConfiguration` ❸ and has been marked as a `@Configuration` ❶ class. This allows you to configure the application in this class by using Spring Java Config. But this example is simple, and there's no additional configuration.

TIP You can add methods annotated with `@Bean` in the configuration class to have Spring automatically call the methods and enlist the returned bean instances into the Spring bean context.

To include the Camel routes, you enable `@ComponentScan` ❷ to scan for classes that have been annotated with `@Component`; these would automatically be enlisted into the Spring bean context. This in turn allows Camel to discover the routes from Spring and automatically include them when Camel starts up.

To make it easy to run this application standalone, you use a `Main` class from camel-spring-javaconfig that in a few lines of code start the application and keep the JVM running ❹ ❺.

How do you unit-test this application? You can test the application using the same `CamelSpringTestSupport` you used when testing Camel with Spring XML files. The difference is in how you create Spring in the `createApplicationContext` method, as shown in the following listing.

Listing 9.6 Testing a Spring Java Config application using CamelSpringTestSupport

```
public class FirstTest extends CamelSpringTestSupport {

  @Override
  protected AbstractApplicationContext createApplicationContext() {
    AnnotationConfigApplicationContext acc = new AnnotationConfigApplicationCon
```

1 Creates a Spring Java Config context

```
    acc.register(MyApplication.class);
```

2 Registers the @Configuration class in the Spring context

```
    return acc;
  }

  @Test
  public void testMoveFile() throws Exception {
```

3 The unit test method is identical with testing using Spring XML files

```
    template.sendBodyAndHeader("file://target/inbox",
            "Hello World", Exchange.FILE_NAME, "hello.txt");

    Thread.sleep(2000);

    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());
    String content = context.getTypeConverter()
                            .convertTo(String.class, target);
    assertEquals("Hello World", content);
```

```
    }
  }
```

The main difference is that Spring Java Config uses `AnnotationConfigApplicationContext` ❶ as the Spring bean container. You then have to configure which `@Configuration` class to use using the `register` method ❷. The rest of the unit test is identical ❸ to listing 9.3, which was testing using Spring XML.

The book's source code contains this example in the chapter9/spring-java-config directory; you can try the example by using the following:

```
mvn test -Dtest=FirstTest
```

You can also unit-test without extending the `CamelSpringTestSupport` class, but instead using a set of annotations. This started becoming more popular recently when JUnit 4.*x* introduced this style.

Unit testing with @RunWith

Instead of extending the `CamelSpringTestSupport` class, you can use the `CamelSpringRunner` as a JUnit runner, which will run the unit tests with Camel support. The following listing shows how to do that.

Listing 9.7 Unit testing with @RunWith instead of extending the CamelSpringTestSupport class

```
@RunWith(CamelSpringRunner.class)    ❶
```

❶ Uses @RunWith with the CamelSpring runner

```
@ContextConfiguration(    ❷
```

❷ Configures Spring to use the @Configuration class

```
    classes = MyApplication.class,    ❷
            loader = CamelSpringDelegatingTestContextLoader.class)    ❷
public class RuntWithFirstTest {

  @Autowired
  private CamelContext context;    ❸
```

❸ Dependency injects CamelContext and ProducerTemplate

```
  @Autowired    ❸
  private ProducerTemplate template;    ❸

  @Test
  public void testMoveFile() throws Exception {    ❹
```

❹ The unit test method is identical to testing without using @RunWith.

```
    template.sendBodyAndHeader("file://target/inbox",
                "Hello World", Exchange.FILE_NAME, "hello.txt");

    Thread.sleep(2000);

    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());
    String content = context.getTypeConverter()
                            .convertTo(String.class, target);
    assertEquals("Hello World", content);
  }
}
```

The unit test uses `CamelSpringRunner` as the `@RunWith` runner ❶. To configure Spring, you use `@ContextConfiguration` ❷; here you refer to the class that holds `@Configuration`. Then the loader refers to `CamelSpringDelegatingTestContextLoader`, which is a class that en-

ables using a set of additional annotations to enable certain features during testing. In this simple example, you're not using any of these features and can omit declaring the loader. But including the loader is a good habit, so you're ready for using Camel-testing capabilities such as auto-mocking and advice. These capabilities are revealed later in this chapter.

Then you dependency-inject `CamelContext` and `ProducerTemplate` ❸, which you use in the unit-test method ❹. You have to do this because the test class no longer extends `CamelTestSpringSupport`, which provides `CamelContext` and `ProducerTemplate` automatically.

You can try this example in the book's source code, in the chapter9/spring-java-config directory, by using the following Maven goal:

```
mvn test -Dtest=RunWithFirstTest
```

What you've learned here about using Spring Java Config is also relevant when using Spring Boot. All Spring Boot applications use the Java configuration style, so let's see how you can build Camel unit tests with Bootify Spring—eh, Spring Boot.

9.2.3 Camel testing with Spring Boot

Spring Boot makes Spring Java Config applications easier to use, develop, run, and test. Using Camel with Spring Boot is easy: you add the following dependency to your Maven pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-boot-starter</artifactId>
  <version>2.20.1</version>
</dependency>
```

And as when using Spring Java Config, your Camel routes can be automatically discovered if you annotate the class with `@Component`:

```
@Component    ❶
```

❶ Annotate the class with @Component to make Spring discover the class easily.

```
public class FileMoveRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("file://target/inbox")
            .to("file://target/outbox");
    }
}
```

Running Camel on Spring Boot is easy. Spring Boot will automatically embed Camel and all its Camel routes (annotated with `@Component`). To set up a Spring Boot application, all you need to do is type the following lines of Java code:

```
@SpringBootApplication    ❶
```

❶ SpringBootApplication annotation to mark this class as a Spring Boot application

```
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);    ❷
    }
}
```

❷ One line to start the Spring Boot application

All you have to do is annotate the class with `@SpringBootApplication` ❶ and write one line of Java code in the `main` method ❷ to run the

application.

Testing this application is as easy as testing the Spring Java Config application covered in the previous section. The only difference is that with Spring Boot, you use `@SpringBootTest` instead of `@ContextConfiguration` to specify which configuration class to use. The following listing shows the Camel unit test with Spring Boot.

Listing 9.8 Unit testing Camel with Spring Boot

```
@RunWith(CamelSpringBootRunner.class)    1
```

1 Uses @RunWith with the CamelSpringBoot runner

```
@SpringBootTest(classes = MyApplication.class)    2
```

2 Configures Spring Boot to use MyApplication as the configuration class

```
public class RunWithFirstTest {

  @Autowired
  private CamelContext context;    3
```

3 Dependency injects CamelContext and ProducerTemplate

```
  @Autowired    3
  private ProducerTemplate template;    3

  @Test
  public void testMoveFile() throws Exception {    4
```

❹

The unit test method is identical to testing without using @RunWith.

```
    template.sendBodyAndHeader("file://target/inbox",
                "Hello World", Exchange.FILE_NAME, "hello.txt");

    Thread.sleep(2000);

    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());
    String content = context.getTypeConverter()
                            .convertTo(String.class, target);
    assertEquals("Hello World", content);
  }
}
```

To run the test as JUnit, you use `CamelSpringBootRunner` as `@RunWith` (notice that the runner is named `Camel SpringBoot Runner`). Then you refer to the Spring Boot application class, which is `MyApplication` ❷. The rest is similar to testing with Spring Java Config ❹, where you must dependency inject `CamelContext` and `ProducerTemplate` ❸ to make them available for use in your test methods.

We prepared this example in the source code in the chapter9/spring-boot directory, and you can try the example by using the following Maven goal:

```
mvn test -Dtest=FirstTest
```

TIP If you want to learn more about Spring Boot, see *Spring Boot in Action* by Craig Walls (Manning, 2016).

This example is a basic example with just one route in one `RouteBuilder` class. What if you have multiple `RouteBuilder` classes? How do you test them?

Testing with only one RouteBuilder class enabled

When you have multiple Camel routes in different `RouteBuilder` classes, Camel on Spring Boot will include all these routes. But you may want to write a unit test that's focused on testing the routes from only a specific `RouteBuilder` class.

Suppose you have two `RouteBuilder` classes named `FooRoute` and `BarRoute`, with related unit-test classes named `FooTest` and `BarTest`. How can you make `FooTest` include only `FooRoute`, and make `BarTest` only include `BarRoute`? Camel makes this easy, as you can use the following patterns to include (or exclude) which `RouteBuilder`s to enable:

- `java-routes-include-pattern`—Used for including `RouteBuilder` classes that match the pattern.
- `java-routes-exclude-pattern`—Used for excluding `RouteBuilder` classes that match the pattern. Exclude takes precedence over include.

You can specify these patterns in your unit-test classes as properties to the `@SpringBootTest` annotation, as highlighted here:

```
@RunWith(CamelSpringBootRunner.class)
@SpringBootTest(classes = {MyApplication.class},
    properties = { "camel.springboot.java-routes-include-pattern=**/Foo*
public class FooTest {
```

In the `FooTest` class, the include pattern is `**/Foo*`, which represents an *ant style* pattern. Notice that the pattern starts with double asterisk, which will match any leading package name. `/Foo*` means the class name must start with `Foo`, such as `FooRoute`, `FoolishRoute`, and so forth.

You can find this example in the chapter9/spring-boot-test-one-route directory, where you can run a test for each route by using the following Maven commands:

```
mvn test -Dtest=FooTest
mvn test -Dtest=BarTest
```

We've covered three ways of testing Camel with Spring. Table 9.2 shows our recommendations for when to use what.

Table 9.2 Recommendations on using Camel with Spring XML, Spring Java Config, or Spring Boot

Spring	Recommendation
Spring XML	The Spring Framework has supported configuring applications using XML files for a long time. This is a well-established and used approach. This is a good choice if you prefer to use the XML DSL for defining your Camel routes.
Spring Java Config	Spring Java Configuration is less known and used even though it's been available for many years. Instead of using Spring Java Config, we recommend using Spring Boot if possible.
Spring Boot	Spring Boot is a popular choice these days and is a recommended approach for running Camel applications. You may think you must use Java code only with Spring Boot, but you can also use Spring XML files. This means you have freedom to define your Camel routes in both Java and XML.

Okay, that was a lot of coverage of testing Camel with Spring. Spring isn't all there is, so let's move on to testing with OSGi Blueprint.

9.2.4 Camel testing with OSGi Blueprint XML

Camel supports defining Camel routes in OSGi Blueprint XML files. In this section, you'll learn how to start writing unit tests for those XML files. But first we need to tell you a story.

THE STORY ABOUT POJOSR THAT BECAME FELIX CONNECTOR

When using OSGi Blueprint with Camel, you must run your Camel applications in an OSGi server such as Apache Karaf/ServiceMix or JBoss Fuse. Many years ago, the only way to test your Camel applications was to deploy them into such an OSGi server.

This changed in 2011 when Karl Pauls created PojoSR, an OSGi Lite container. PojoSR runs without an OSGi framework and uses a flat class loader. Instead, PojoSR bootstraps a simulated OSGi environment: you colocate your unit test and Camel dependencies and run in the same flat class loader. In other words, it runs in the same JVM locally and therefore starts up fast. You'll also be pleased to know that because it all runs in the same JVM, Java debugging is much easier: set a breakpoint in your IDE editor and click the debug button—no need for remote Java debugging. But the flip side is that the more OSGi needs you have, the bigger the chance that PojoSR has reached its limits. When you have such needs, you can turn to using an alternative testing framework called Pax Exam or Arquillian (covered in section 9.6). PojoSR has since become part of Apache Felix as the Felix Connect project.

To start unit-testing Camel with OSGi Blueprint XML files, you add the following dependency to your Maven pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test-blueprint</artifactId>
  <version>2.20.1</version>
</dependency>
```

To keep this example basic, you'll use the following OSGi Blueprint XML file with the file copier Camel route, as shown in the following listing.

Listing 9.9 An OSGi Blueprint XML file with a basic Camel route (firststep.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
      https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
      <route>
        <from uri="file://target/inbox"/>
        <to uri="file://target/outbox"/>
      </route>
    </camelContext>
  </blueprint>
```

To unit test this route, you extend the class `CamelBlueprintTestSupport` and write the unit test almost like that in the previous section. The following listing shows the code.

Listing 9.10 OSGi Blueprint-based unit test

```
public class BlueprintFirstTest extends CamelBlueprintTestSupport {   (1)
```

(1) Test class must extend CamelBlueprintTestSupport

```
  @Override
  protected String getBlueprintDescriptor() {
    return "camelinaction/firststep.xml";   (2)
```

(2) Loads the OSGi Blueprint XML file

```
  }

  @Test
  public void testMoveFile() throws Exception {   (3)
```

❸

Test method is similar to previous examples

```
      template.sendBodyAndHeader("file://target/inbox",
                  "Hello World", Exchange.FILE_NAME, "hello.txt");

      Thread.sleep(2000);

      File target = new File("target/outbox/hello.txt");
      assertTrue("File not moved", target.exists());
      String content = context.getTypeConverter()
                              .convertTo(String.class, target);
      assertEquals("Hello World", content);
    }
  }
```

As you can see, the `BlueprintFirstTest` class extends `CamelBlueprintTestSupport`, which is required ❶. Then you override the method `getBlueprintDescriptor`, where you return the location in the classpath of the OSGi Blueprint XML file ❷ (you can specify multiple files by using a comma to separate the locations). The rest of the test method is similar to previous examples ❸.

You can try this example, located in the chapter9/blueprint-xml directory, using the following Maven goal:

```
mvn test -Dtest=BlueprintFirstTest
```

We have one more trick to show you with OSGi Blueprint. When using OSGi, you can compose your applications into separate bundles and services. You may compose shared services or let different teams build different services. Then, all together, these services can be installed in the same OSGi servers and just work together—famous last words.

Mocking OSGi Services

If you're on one of these teams, how can you unit-test your Camel applications with shared services across your Camel application? What you can

do with camel-test-blueprint is mock these services and register them manually in the OSGi service registry.

Suppose your Camel Blueprint route depends on an OSGi service, as shown here:

```
<reference id="auditService" interface="camelinaction.AuditService"/>   ❶
```

❶ Reference to existing OSGi service

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  <route>
    <from uri="file://target/inbox"/>
    <bean ref="auditService"/>   ❷
```

❷ Calling the OSGi service from Camel route

```
    <to uri="file://target/outbox"/>
  </route>
</camelContext>
```

The `<reference>` ❶ is a reference to an existing OSGi service in the OSGi service registry that's identified only by the interface that must be of type `camelinaction.AuditService`. The service is then used in the Camel route ❷.

If you attempt to unit-test this Camel route with camel-test-blueprint, the test will fail with an OSGi Blueprint error stating a time-out error waiting for the service to be available.

Yes, this is a simple example, but imagine that your Camel applications are using any number of OSGi services that are built by other teams. How can you build and run your Camel unit tests?

What you can do is implement mocks for these services and register these fake services in your unit test. For example, you could implement a mocked service that when called will send the Camel `Exchange` to a mock endpoint, as shown here:

```
public class MockAuditService implements AuditService {
  public void audit(Exchange exchange) {
      exchange.getContext().createProducerTemplate()
          .send("mock:audit", exchange);
  }
}
```

Then you can register `MockAuditService` as the fake service in the OSGi service registry as part of your unit test using the `addServicesOnStartup` method:

```
protected void addServicesOnStartup(Map<String,
                            KeyValueHolder<Object, Dictionary>> services)
  MockAuditService mock = new MockAuditService()
```

Creates mock of the AuditService

```
  services.put(AuditService.class.getName(), asService(mock, null));
```

Registers the mock into the OSGi ServiceRegistry

```
}
```

You can find this example with the source code in the chapter9/blueprint-xml-service directory, and you can try the example by using the following Maven goal:

```
mvn test
```

We'll now move on to using Camel with CDI. You'll see how to start testing Camel running in a CDI container such as JBoss Weld and then using a

Java EE application server such as WildFly.

9.2.5 Camel testing with CDI

Camel has great integration with Contexts and Dependency Injection (CDI) that allows you to build Camel routes using Java code.

TIP Chapter 7 covered CDI. You'll encounter CDI again in chapter 15.

To start unit-testing Camel with CDI, you add the following dependency to your Maven pom.xml file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-test-cdi</artifactId>
  <version>2.20.1</version>
</dependency>
```

As usual we'll use a basic Camel example to show you the ropes to get on board testing with CDI. The Camel route used for testing is a file copier example:

```
@Singleton  1
```

Annotates the class with @Singleton to make Camel discover the class easily

---

```
public class FileMoveRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {
    from("file:target/inbox")
      .to("file:target/outbox");
  }
}
```

The class is annotated with `@Singleton`, which allows Camel to discover the instance on startup and automatically include the route.

To unit-test this Camel route, you don't extend a base class but use `CamelCdiRunner`, a JUnit runner, as shown next.

Listing 9.11 Unit testing Camel CDI application

```
@RunWith(CamelCdiRunner.class)    ❶
```

❶ Uses @RunWith with the Camel CDI runner

```
public class FirstTest {

  @Inject
  private CamelContext context;    ❷
```

❷ Dependency injects CamelContext and ProducerTemplate

```
  @Inject @Uri("file:target/inbox")
  private ProducerTemplate template;    ❷

  @Test
  public void testMoveFile() throws Exception {    ❸
```

❸ The unit test method is identical to previous unit tests covered previously.

```
    template.sendBodyAndHeader("Hello World",
                               Exchange.FILE_NAME, "hello.txt");

    Thread.sleep(2000);

    File target = new File("target/outbox/hello.txt");
    assertTrue("File not moved", target.exists());
```

```
    String content = context.getTypeConverter()
                            .convertTo(String.class, target);
    assertEquals("Hello World", content);
  }

}
```

To run the test as JUnit, you use `CamelCdiRunner` as `@RunWith` ❶. Because the `FirstTest` class doesn't extend a base class, you need to dependency-inject the resources used during testing such as `CamelContext` and `ProducerTemplate` ❷. The test method is implemented similarly to what's been covered before in this section ❸.

You can try this example, which is provided with the source code in the chapter9/cdi directory, using the following Maven goal:

```
mvn test -Dtest=FirstTest
```

The JBoss Weld CDI container doesn't offer much power besides being able to run a CDI application. If you're looking for a more powerful and better container, take a look at WildFly Swarm.

9.2.6 Camel testing with WildFly Swarm

You first encountered WildFly Swarm in this book in chapter 7. You may think WildFly Swarm is similar to Spring Boot but for the Java EE specification. Therefore, it's best to run Camel with CDI on WildFly Swarm. This time, you'll use a simple Camel route that routes a message between two in-memory SEDA queues, as shown here:

```
@Singleton
```

Annotates the class with @Singleton to make CDI discover the class easily

```
public class SedaRoute extends RouteBuilder {
  @Override
  public void configure() throws Exception {
    from("seda:inbox")
      .to("seda:outbox");
```

```
    }
}
```

WildFly Swarm comes with powerful test capabilities using Arquillian. Setting up Arquillian is fairly trivial. First, you need to add the WildFly Swarm Arquillian dependency to the Maven pom.xml file:

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>arquillian</artifactId>
  <scope>test</scope>
</dependency>
```

Then all that's left is to write the unit test, shown in the following listing.

Listing 9.12 Testing Camel using Arquillian on WildFly Swarm

```
@RunWith(Arquillian.class)    (1)
```

(1) Sets up unit test to run with Arquillian

```
@DefaultDeployment(type = DefaultDeployment.Type.JAR)    (2)
```

(2) Uses the default deployment

```
public class WildFlySwarmCamelTest {

  @Inject    (3)
```

(3) Injects CamelContext

```
  private CamelContext camelContext;    (3)
```

```
  @Inject @Uri("seda:inbox")    ❹
```

❹ Injects ProducerTemplate to send to seda:inbox

```
  private ProducerTemplate template;    ❹

  @Test
  public void testSeda() throws Exception {
    template.sendBody("Hello Swarm");

    ConsumerTemplate consumer = camelContext.createConsumerTemplate();
    Object body = consumer.receiveBody("seda:outbox", 5000);    ❺
```

❺ Receives message from seda:outbox

```
    assertEquals("Hello Swarm", body);
  }
}
```

The class `WildFlySwarmCamelTest` has been annotated with JUnit `@RunWith(Arquillian)` ❶ to set up and run the unit test using Arquillian. Because your Camel application is running inside WildFly Swarm, you've annotated the class with `@DefaultDeployment` ❷, which instructs Arquillian to include your entire application together with WildFly Swarm. Notice that you specify `type = DefaultDeployment.Type.JAR`, which is required when using WildFly Swarm in JAR packaging mode, instead of the default WAR packaging mode. You'll do this most often with WildFly Swarm because it's intended for microservices, where you deploy and run only one application in the server. But Arquillian is capable of deploying only what you've instructed when you run in traditional application servers, which you'll try in the following section.

The test uses CDI to inject `CamelContext` ❸ and `ProducerTemplate` ❹, which you use in the test method. The producer is used to send a message

to the `seda:inbox` endpoint followed by the `ConsumerTemplate` ❺ to retrieve the message from the `seda:outbox` endpoint.

The consumer is using a five-second time-out. If you don't specify a time-out, the consumer will wait until a message is received. Because this is a unit test, you want to fail instead if no message is routed by Camel. This will happen in case of a time-out because the message body would be `null` and fail the equals check at the end of the test.

You can try this example, which is part of the source code in the chapter9/wildfly-swarm directory, by using the following Maven goal:

```
mvn test -Dtest=WildFlySwarmCamelTest
```

You can also run a Camel application in Java EE application servers such as WildFly. But how do you test with Camel and WildFly?

9.2.7 Camel testing with WildFly

There are several ways of testing Camel with WildFly. The simplest form is a regular unit test that runs standalone outside WildFly. This unit test would be much the same as those we've already covered:

- Testing Camel with Java routes (covered in section 9.1.2)
- Testing Camel with Spring XML routes (covered in section 9.2.1)

In this section, we raise the bar to talk about how to test Camel with WildFly when the test runs inside the WildFly application server. This allows you to do integration tests with *the real thing* (the application server).

You'll build a simple example that uses Java EE technology together with Camel running in a WildFly application server. The example exposes a servlet that calls a Camel route and returns a response. Figure 9.3 illustrates how the example is deployed as a WAR deployment running inside the WildFly server.

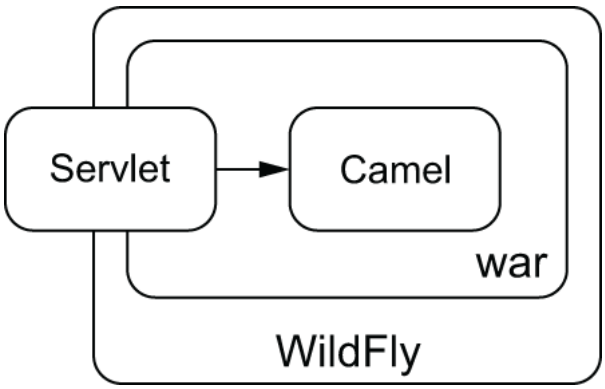

Figure 9.3 A WAR deployment using servlet and Camel running in WildFly application server

The servlet is implemented as shown in the following listing.

Listing 9.13 A hello Servlet that calls a Camel route

```
@WebServlet(name = "HelloServlet", urlPatterns = {"/*"}, loadOnStartup =
public class HelloServlet extends HttpServlet {   ❶
```

❶ @WebServlet to make this class act as a servlet

```
  @Inject
  private ProducerTemplate producer;   ❷
```

❷ Dependency injects a Camel ProducerTemplate using CDI injection

```
  protected void doGet(HttpServletRequest req,
                       HttpServletResponse res) throws IOException {
    String name = req.getParameter("name");

    String s = producer.requestBody("direct:hello", name, String.class);   ❸
```

❸ Calls the Camel route to retrieve an output to use as response

```
      res.getOutputStream().print(s);
    }
  }
```

The servlet is implemented in the `HelloServlet` class. By annotating the class with `@WebServlet` ❶, you don't have to register the servlet by using the WEB-INF/web.xml file. The servlet integrates with Camel by using CDI to inject `ProducerTemplate` ❷. The servlet implements the `doGet` method that handles HTTP `GET` methods. In this method, the query parameter name is extracted and used as message body when calling the Camel route ❸ to obtain a response message from Camel.

The Camel route is a simple route that transforms a message:

```
@ContextName("helloCamel")
public class HelloRoute extends RouteBuilder {

  public void configure() throws Exception {
    from("direct:hello")
        .transform().simple("Hello ${body}");
  }
}
```

This example is packaged as a WAR deployment unit that can be deployed to any Java EE web server such as Apache Tomcat or WildFly. We'll now show you how to test your deployments running inside WildFly.

To be able to test this, you use Arquillian, which allows you to write unit or integration tests that can deploy your application and tests together and run the tests inside a running WildFly server:

Arquillian provides a component model for integration tests, which includes dependency injection and container lifecycle management. Instead of managing a runtime in your test, Arquillian brings your test to the runtime.—Arquillian website, http://arquillian.org/modules/core-platform/

Testing WildFly using Arquillian

To use Arquillian to perform integration tests using WildFly, you need to add the `arquillian-bom` and `shrinkwrap-resolver-bom` BOMs to your

Maven pom.xml file. In addition, you need to add the following three dependencies:

- arquillian-junit-container—Provides the Arquillian JUnit runner
- shrinkwrap-resolver-impl-maven—To resolve dependencies using Maven
- wildfly-arquillian-container-managed—Deploys and runs the tests in WildFly

You can find the exact details in the pom.xml file in the example source code, located in the chapter9/wildfly directory.

Writing an Arquillian-based integration test is similar to writing a unit test. The following listing shows what it takes.

Listing 9.14 System test that tests your servlet and Camel application running in WildFly

```
@RunWith(Arquillian.class)    (1)
```

(1) JUnit runner to run using Arquillian

```
public class FirstWildFlyIT {

  @Inject
  CamelContext camelContext;    (2)
```

(2) Dependency injects CamelContext

```
  @Deployment
  public static WebArchive createDeployment() {
    File[] files = Maven.resolver()
                    .loadPomFromFile("pom.xml")
```

```
                        .importRuntimeDependencies()
            .resolve().withTransitivity().asFile();  (3)
```

(3) ShrinkWrap to resolve all the runtime dependencies from the Maven pom.xml file

```
    WebArchive archive = ShrinkWrap.create(WebArchive.class,  (4)
```

(4) Creates WebArchive deployment using ShrinkWrap

```
                                                    "mycamel-wildfly.war");
      archive.addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml");
    archive.addPackages(true, "camelinaction");  (5)
```

(5) Includes the project classes

```
    archive.addAsLibraries(files);  (6)
```

(6) Adds all the Maven dependencies as libraries in the deployment unit

```
      return archive;
    }

    @Test
  public void testHello() throws Exception {  (7)
```

(7) The test method that calls the Camel route

```
    String out = camelContext.createProducerTemplate()
                    .requestBody("direct:hello", "Donald", String.class
    assertEquals("Hello Donald", out);
  }
}
```

The test class `FirstWildFlyIT` uses the Arquillian JUnit runner ❶ that lets Arquillian be in charge of the tests. Then you use CDI to inject `CamelContext` into the test class ❷. To include all needed dependencies in the deployment unit, you use ShrinkWrap Maven resolver ❸ that includes all the runtime dependencies and their transitive dependencies. The deployment unit is then created ❹ by using ShrinkWrap as a WAR deployment with the name `mycamel-wildly.war`. It's important that you use the correct name of the WAR file as configured in the Maven pom.xml file:

```
<finalName>mycamel-widlfly</finalName>
```

The method `addPackages` ❺ includes all the classes from the root package `camelinaction` and all subpackages. This ensures that all the classes of this project are included in the deployment unit. The method `addAsLibraries` ❻ adds all the dependencies ❸ as JARs to the deployment unit. The unit-test method is a simple test that calls the Camel route ❼. You could've chosen to perform an HTTP call to the servlet instead, but then you'd need to add an HTTP library. You cheat a bit and just call the Camel route.

Okay, that's a mouthful to take in; are you ready for testing? No, you still have two tasks to do: set up Arquillian and install WildFly.

You configure Arquillian in the arquillian.xml file, which you put in the src/test/resources directory. Its content is shown here:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<arquillian xmlns="http://jboss.org/schema/arquillian"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="
              http://jboss.org/schema/arquillian
              http://jboss.org/schema/arquillian/arquillian_1_0.xsd">
```

```
    <engine>
      <property name="deploymentExportPath">target</property>    ❶
```

❶ To use the target directory during deployment and testing

```
    </engine>
    <container qualifier="managed" default="true">    ❷
```

❷ Uses a managed WildFly container

```
      <configuration>
        <property name="jbossHome">${env.JBOSS_HOME}</property>    ❸
```

❸ The location of the JBoss WildFly installation

```
        <property name="serverConfig">standalone.xml</property>    ❹
```

❹ To use the standalone.xml configuration with WildFly

```
        <property name="allowConnectingToRunningServer">true</property>
      </configuration>
    </container>
  </arquillian>
```

The arquillian.xml configuration file is fairly easy to comprehend. At first, you configure to use the target directory ❶ as a work directory so it's easy to clean up using a Maven clean goal. You use Arquillian container testing in managed mode ❷. This means an existing installation of WildFly is used, so you need to configure the directory where WildFly is installed. The recommended way is to use the `JBOSS_HOME` environment

variable ❸. Then you tell Arquillian to use the default WildFly standalone.xml configuration file when running WildFly ❹.

TIP What you learn here about using Arquillian for testing with WildFly is similar in principle to using other containers such as Apache Tomcat or Jetty.

The last piece of the puzzle is to install WildFly.

Installing WildFly

You can download WildFly from http://wildfly.org. The installation of WildFly is a matter of unzipping the downloaded file and setting up the `JBOSS_HOME` environment variable.

We have installed WildFly 11 in the /opt directory and set up the `JBOSS_HOME` environment as follows:

```
export JBOSS_HOME=/opt/wildfly-11.0.0.Final
```

After this is done, you're ready for running the integration tests, which is as simple as running a Maven test. We've included this example in the source code in the chapter9/wildfly directory, and you can try the example by using the following:

```
mvn integration-test -P wildfly
```

We took our time to walk you through setting up a project for system tests with Arquillian that runs in WildFly. What you've learned here will come in handy in section 9.6, which provides more information about system tests using various test libraries such as Arquillian and Pax Exam.

You've now seen the Camel Test Kit and learned to use its JUnit extension to write your first unit tests with various runtime platforms such as standalone Java, Spring Boot, OSGi, CDI, and WildFly. Camel helps a lot when working with files, but things get more complex when you use more protocols—especially complex ones such as Java Message Service

(JMS) messaging. Testing an application that uses many protocols has always been challenging.

This is why mocks were invented. Using mocks, you can simulate real components and reduce the number of variables in your tests. Mock components are the topic of the next section.

Now is a good time to take a little break. We're changing the scene from running Camel tests on various runtimes to using the Camel mock component, which is useful in your unit tests.

9.3 Using the mock component

The *mock component* is a cornerstone when testing with Camel; it makes testing much easier. In much the same way that a car designer uses a crash-test dummy to simulate vehicle impact on humans, the mock component is used to simulate real components in a controlled way.

Mock components are useful in several situations:

- When the real component doesn't yet exist or isn't reachable in the development and test phases. For example, if you have access to the component in only preproduction and production phases.
- When the real component is slow or requires much effort to set up and initialize, such as a database.
- When you'd have to incorporate special logic into the real component for testing purposes, which isn't practical or possible.
- When the component returns nondeterministic results, such as the current time, which would make it difficult to unit-test at any given time of day.
- When you need to simulate errors caused by network problems or faults from the real component.

Without the mock component, your only option is to test using the real component, which is usually much harder. You may already have used mocking before—many frameworks blend in well with testing frameworks such as JUnit.

Camel takes testing seriously, and the mock component was included in the first release of Camel. The fact that it resides in the camel-core JAR in-

dicates its importance. The mock component is used rigorously in unit-testing Camel itself.

In this section, you'll look at how to use the mock component in unit tests and how to add mocking to existing unit tests. Then you'll spend some time using mocks to set expectations to verify test results, as this is where the mock component excels.

Let's get started.

9.3.1 Introducing the mock component

Figure 9.4 illustrates the three basic steps of testing.

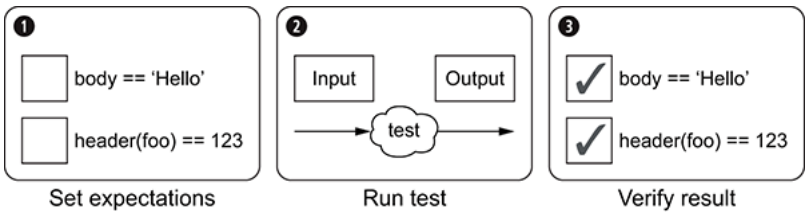

Figure 9.4 Three steps for testing: set expectations, run the test, and verify the result

Before the test is started, you set the expectations of what should happen ❶. Then you run the test ❷. Finally, you verify the outcome of the test against the expectations ❸. The Camel mock component allows you to easily implement these steps when testing Camel applications. On the mock endpoints, you can set expectations that are used to verify the test results when the test completes.

Mock components can verify a rich variety of expectations, such as the following:

- The correct number of messages are received on each endpoint.
- The messages arrive in the correct order.
- The correct payloads are received.
- The test ran within the expected time period.

Mock components allow you to configure coarse- and fine-grained expectations and simulate errors such as network failures. Let's get started and try using the mock component.

9.3.2 Unit-testing with the mock component

As you learn how to use the mock component, you'll use the following basic route to keep things simple:

```
from("jms:topic:quote").to("mock:quote");
```

This route will consume messages from a JMS topic, named `quote`, and route the messages to a mock endpoint with the name `quote`.

The mock endpoint is implemented in Camel as the `org.apache.camel.component.mock.MockEndpoint` class; it provides a large number of methods for setting expectations. Table 9.3 lists the most commonly used methods on the mock endpoint.

Table 9.3 Commonly used methods in the MockEndpoint class

Method	Description
`expectedMessageCount(int count)`	Specifies the expected number of messages arriving at the endpoint
`expectedMinimumMessageCount (int count)`	Specifies the expected minimum number of messages arriving at the endpoint
`expectedBodiesReceived (Object... bodies)`	Specifies the expected message bodies and their order arriving at the endpoint
`expectedBodiesReceivedInAnyOrder (Object... bodies)`	Specifies the expected message bodies arriving at the endpoint—ordering doesn't matter
`assertIsSatisfied ()`	Validates that all expectations set on the endpoint are satisfied

The `expectedMessageCount` method is exactly what you need to set the expectation that one message should arrive at the `mock:quote` endpoint. You can do this as shown in the following listing.

Listing 9.15 Using `MockEndpoint` in unit testing

```
public class FirstMockTest extends CamelTestSupport {

  protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
      public void configure() throws Exception {
        from("jms:topic:quote").to("mock:quote");
```

```
        }
      };
    }

    @Test
    public void testQuote() throws Exception {
      MockEndpoint quote = getMockEndpoint("mock:quote");
    quote.expectedMessageCount(1);  ❶
```

---

❶

Expects one message

```
      template.sendBody("jms:topic:quote", "Camel rocks");

    quote.assertIsSatisfied();  ❷
```

---

❷

Verifies expectations

```
    }
  }
```

To obtain the `MockEndpoint`, you use the `getMockEndpoint` method from the `CamelTestSupport` class. Then you set your expectations—in this case, you expect one message to arrive ❶. You start the test by sending a message to the JMS topic, and the mock endpoint verifies whether the expectations were met by using the `assertIsSatisfied` method ❷. If a single expectation fails, Camel throws a `java.lang.AssertionError` stating the failure.

You can compare what happens in listing 9.14 to what you saw in figure 9.4: you set expectations, run the test, and verify the results. It can't get any simpler than that.

NOTE By default, the `assertIsSatisfied` method runs for 10 seconds before timing out. You can change the wait time with the `setResultWaitTime(long timeInMillis)` method if you have unit tests that run for a long time. But it's easier to specify the time-out directly as a parameter to the assert method: `assertIsSatisfied(5, TimeUnit.SECONDS);`.

Replacing JMS with Stub

Listing 9.14 uses JMS, but for now let's keep things simple by using the stub component to simulate JMS. (You'll look at testing JMS with ActiveMQ in section 9.4.) The stub component is essentially the SEDA component with parameter validation turned off. This makes it possible to stub any Camel endpoint by prefixing the endpoint with `stub:`. In this example, you have the endpoint `jms:topic:quote`, which can be stubbed as `stub:jms:topic:quote`. The route will be as simple as this:

```
from("stub:jms:topic:quote").to("mock:quote");
```

The producer can then send a message to the queue by using the following:

```
template.sendBody("stub:jms:topic:quote", "Camel rocks");
```

We've provided this example in the source code, in the chapter9/mock directory; you can try the example by using the following Maven goals:

```
mvn test -Dtest=FirstMockTest
mvn test -Dtest=SpringFirstMockTest
```

You may have noticed in listing 9.15 that the expectation is coarse-grained in the sense that you expect only one message to arrive. You don't specify anything about the message's content or other characteristics, so you don't know whether the message that arrives is the same "Camel rocks" message that was sent. The next section covers how to test this.

9.3.3 Verifying that the correct message arrives

The `expectedMessageCount` method can be used to verify only that a certain number of messages arrive. It doesn't dictate anything about the content of the message. Let's improve the unit test in listing 9.15 so that it expects the message being sent to match the message that arrives at the mock endpoint.

You can do this by using the `expectedBodiesReceived` method, as follows:

```
@Test
public void testQuote() throws Exception {
  MockEndpoint mock = getMockEndpoint("mock:quote");
  mock.expectedBodiesReceived("Camel rocks");

  template.sendBody("stub:jms:topic:quote", "Camel rocks");

  mock.assertIsSatisfied();
}
```

This is intuitive and easy to understand, but the method states *bodies*, plural, as if there could be more bodies. Camel does support expectations of multiple messages, so you could send in two messages. Here's a revised version of the test:

```
@Test
public void testQuotes() throws Exception {
  MockEndpoint mock = getMockEndpoint("mock:quote");
  mock.expectedBodiesReceived("Camel rocks", "Hello Camel");

  template.sendBody("stub:jms:topic:quote", "Camel rocks");
  template.sendBody("stub:jms:topic:quote", "Hello Camel");

  mock.assertIsSatisfied();
}
```

Camel now expects two messages to arrive in the specified order. Camel will fail the test if the "Hello Camel" message arrives before the "Camel rocks" message.

If the order doesn't matter, you can use the `expectedBodiesReceivedInAnyOrder` method instead, like this:

```
mock.expectedBodiesReceivedInAnyOrder("Camel rocks", "Hello Camel");
```

It could hardly be any easier than that.

But if you expect a much larger number of messages to arrive, the number of bodies you pass in as an argument will be large. How can you do that? The answer is to use a `List` containing the expected bodies as a parameter:

```
List bodies = ...
mock.expectedBodiesReceived(bodies);
```

This example is available in the source code, in the chapter9/mock directory; you can try the example by using the following Maven goals:

```
mvn test -Dtest=FirstMockTest
mvn test -Dtest=SpringFirstMockTest
```

The mock component has many other features. Let's continue by exploring how to use expressions to set fine-grained expectations.

9.3.4 Using expressions with mocks

Suppose you want to set an expectation that a message should contain the word *Camel* in its content. The following listing shows one way of doing this.

Listing 9.16 Using expressions with `MockEndpoint` to set expectations

```
@Test
public void testIsCamelMessage() throws Exception {
  MockEndpoint mock = getMockEndpoint("mock:quote");
 mock.expectedMessageCount(2);    ❶
```

❶ Expects two messages

```
    template.sendBody("stub:jms:topic:quote", "Hello Camel");
    template.sendBody("stub:jms:topic:quote", "Camel rocks");

    assertMockEndpointsSatisfied();    ❷
```

---

❷

Verifies two messages received

```
    List<Exchange> list = mock.getReceivedExchanges();    ❸
```

---

❸

Verifies “Camel” is in received messages

```
    String body1 = list.get(0).getIn().getBody(String.class);    ❸
    String body2 = list.get(1).getIn().getBody(String.class);    ❸
    assertTrue(body1.contains("Camel"));    ❸
    assertTrue(body2.contains("Camel"));    ❸
}
```

First, you set up your expectation that the `mock:quote` endpoint will receive two messages ❶. You then send two messages to the JMS topic to start the test. Next, you assert that the mock received the two messages by using the `assertMockEndpointsSatisfied` method ❷, which is a one-stop method for asserting all mocks. This method is more convenient to use than having to invoke the `assertIsSatisfied` method on *every* mock endpoint you may have in use.

At this point, you can use the `getReceivedExchanges` method to access all the exchanges the `mock:quote` endpoint has received ❸. You use this method to get hold of the two received message bodies so you can assert that they contain the word *Camel*.

At first you may think it a bit odd to define expectations in two places—before and after the test has run. Is it not possible to define the expecta-

tions in one place, such as before you run the test? Yes, it is, and this is where Camel expressions come into the game.

NOTE The `getReceivedExchanges` method still has its merits. It allows you to work with the exchanges directly, giving you the ability to do whatever you want with them.

Table 9.4 lists some additional `MockEndpoint` methods that let you use expressions to set expectations.

Table 9.4 Expression-based methods commonly used on MockEndpoint

Method	Description
`message (int index)`	Defines an expectation on the *n*th message received
`allMessages()`	Defines an expectation on all messages received
`expectsAscending (Expression expression)`	Expects messages to arrive in ascending order
`expectsDescending (Expression expression)`	Expects messages to arrive in descending order
`expectsDuplicates (Expression expression)`	Expects duplicate messages
`expectsNoDuplicates (Expression expression)`	Expects no duplicate messages
`expects (Runable runable)`	Defines a custom expectation

You can use the `message` method to improve the unit test in listing 9.14 and group all your expectations together, as shown here:

```
@Test
public void testIsCamelMessage() throws Exception {
  MockEndpoint mock = getMockEndpoint("mock:quote");
  mock.expectedMessageCount(2);
  mock.message(0).body().contains("Camel");
  mock.message(1).body().contains("Camel");

  template.sendBody("stub:jms:topic:quote", "Hello Camel");
  template.sendBody("stub:jms:topic:quote", "Camel rocks");

  assertMockEndpointsSatisfied();
}
```

Notice that you can use the `message(int index)` method to set an expectation that the body of the message should contain the word *Camel*. Instead of doing this for each message based on its index, you can use the `allMessages` method to set the same expectation for all messages:

```
mock.allMessages().body().contains("Camel");
```

So far, you've seen expectations based on only the message body, but what if you want to set an expectation based on a header? That's easy—you use `header(name)`, as follows:

```
mock.message(0).header("JMSPriority").isEqualTo(4);
```

You probably noticed the `contains` and `isEqualTo` methods used in the preceding couple of code snippets. They're builder methods used to create predicates for expectations. Table 9.5 lists all the builder methods available.

Table 9.5 Builder methods for creating predicates to be used as expectations

Method	Description
`contains (Object value)`	Sets an expectation that the message body contains the given value.
`isInstanceOf (Class type)`	Sets an expectation that the message body is an instance of the given type.
`startsWith (Object value)`	Sets an expectation that the message body starts with the given value.
`endsWith (Object value)`	Sets an expectation that the message body ends with the given value.
`in (Object... values)`	Sets an expectation that the message body is equal to any of the given values.
`isEqualTo (Object value)`	Sets an expectation that the message body is equal to the given value.
`isNotEqualTo (Object value)`	Sets an expectation that the message body isn't equal to the given value.
`isGreaterThan (Object value)`	Sets an expectation that the message body is

Method	Description
	greater than the given value.
`isGreaterThanOrEqual(Object value)`	Sets an expectation that the message body is greater than or equal to the given value.
`isLessThan(Object value)`	Sets an expectation that the message body is less than the given value.
`isLessThanOrEqual(Object value)`	Sets an expectation that the message body is less than or equal to the given value.
`isNull(Object value)`	Sets an expectation that the message body is `null`.
`isNotNull(Object value)`	Sets an expectation that the message body isn't `null`.
`matches(Expression expression)`	Sets an expectation that the message body matches the given expression. The expression can be any of the supported Camel expression languages such as Simple, Groovy, SpEL, or MVEL.

Method	Description
`regex(String pattern)`	Sets an expectation that the message body matches the given regular expression.

At first, it may seem odd that the methods in table 9.5 often use `Object` as the parameter type. Why not a specialized type such as `String`? That's because of Camel's strong type-converter mechanism, which allows you to compare apples to oranges; Camel can regard both of them as fruit and evaluate them accordingly. You can compare strings with numeric values without having to worry about type mismatches, as illustrated by the following two code lines:

```
mock.message(0).header("JMSPriority").isEqualTo(4);
mock.message(0).header("JMSPriority").isEqualTo("4");
```

Now suppose you want to create an expectation that all messages contain the word *Camel* and end with a period. You could use a regular expression to set this in a single expectation:

```
mock.allMessages().body().regex(".*Camel.*\\.");
```

This will work, but Camel allows you to enter multiple expectations, so instead of using the `regex` method, you can create a more readable solution:

```
mock.allMessages().body().contains("Camel");
mock.allMessages().body().endsWith(".");
```

The source code includes examples that you can try by running the following Maven command from the chapter9/mock directory:

```
mvn test -Dtest=SecondMockTest
mvn test -Dtest=SpringSecondMockTest
```

You've learned a lot about how to set expectations, including fine-grained ones using the builder methods listed in table 9.5.

Now let's get back to the mock components and learn about using mocks to simulate real components. This is useful when the real component isn't available or isn't reachable from a local or test environment.

9.3.5 Using mocks to simulate real components

Suppose you have a route like the following one, in which you expose an HTTP service using Jetty so clients can obtain an order status:

```
from("jetty:http://web.rider.com/service/order")
  .process(new OrderQueryProcessor())
  .to("netty4:tcp://miranda.rider.com:8123?textline=true")
  .process(new OrderResponseProcessor());
```

Clients send an HTTP `GET`, with the order ID as a query parameter, to the http://web.rider.com/service/order URL. Camel will use the `OrderQueryProcessor` to transform the message into a format that the Rider Auto Parts mainframe (named Miranda) understands. The message is then sent to Miranda using TCP, and Camel waits for the reply to come back. The reply message is then processed using the `OrderResponseProcessor` before it's returned to the HTTP client.

Now suppose you're asked to write a unit test to verify that clients can obtain the order status. The challenge is that you don't have access to Miranda, which contains the order status. You're asked to simulate this server by replying with a canned response.

Camel provides the two methods listed in table 9.6 to help simulate a real component.

Table 9.6 Methods to control responses when simulating a real component

Method	Description
`whenAnyExchangeReceived(Processor processor)`	Uses a custom processor to set a canned reply
`whenExchangeReceived(int index, Processor processor)`	Uses a custom processor to set a canned reply when the *n*th message is received

You can simulate a real endpoint by mocking it with the mock component and using the methods in table 9.6 to control the reply. To do this, you need to replace the endpoint in the route with the mocked endpoint, which is done by replacing it with `mock:miranda`. Because you want to run the unit test locally, you also need to change the HTTP hostname to `localhost`, allowing you to run the test locally on your own laptop:

```
from("jetty:http://localhost:9080/service/order")
  .process(new OrderQueryProcessor())
  .to("mock:miranda")
  .process(new OrderResponseProcessor());
```

The unit test that uses the preceding route follows.

Listing 9.17 Simulating a real component by using a mock endpoint

```
public class MirandaTest extends CamelTestSupport {

  @Test
  public void testMiranda() throws Exception {
    MockEndpoint mock = getMockEndpoint("mock:miranda");
    mock.expectedBodiesReceived("ID=123");
    mock.whenAnyExchangeReceived(e ->
   e.getIn().setBody("ID=123,STATUS=IN PROGRESS")   (1)
```

(1) Returns canned response

```
    );

    String url = "http://localhost:9080/service/order?id=123";

    String out = template.requestBody(url, null, String.class);
  assertEquals("IN PROGRESS", out);   ❷
```

❷

Verifies expected reply

```
    assertMockEndpointsSatisfied();
  }

  protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
      public void configure() throws Exception {
        from("jetty://http://localhost:9080/service/order")
      .transform().message(m -> "ID=" + m.getHeader("id"))   ❸
```

❸

Transforms to format understood by Miranda

```
          .to("mock:miranda")
          .transform().body(String.class,
              b -> StringHelper.after(b, "STATUS="));   ❹
```

❹

Transforms to response format

```
      }
    };
  }
}
```

In the `testMiranda` method, you obtain the `mock:miranda` endpoint, which is the mock that simulates the Miranda server, and you set an expectation that the input message contains the body `"ID=123"`. To return a canned reply, you use the `whenAnyExchangeReceived` method ❶, which allows you to use a custom processor to set the canned response. This response is set to be `ID=123,STATUS=IN PROGRESS`.

Then you start the unit test by sending a message to the http://localhost:9080/service/order?id=123 endpoint; the message is an HTTP `GET` using the `requestBody` method from the `template` instance. You then assert that the reply is `IN PROGRESS` by using the regular JUnit `assertEquals` method ❷.

Instead of using two processors to transform the data to and from the format that the Miranda server understands, you're using Camel's Java 8 DSL to perform the message transformations ❸ ❹ in as few lines of code as possible.

You can find the code for this example in the chapter9/miranda folder of the book's source code, and you can try the example by using the following Maven goal:

```
mvn test -Dtest=MirandaTest
mvn test -Dtest=MirandaJava8Test
```

You've now learned all about the Camel Test Kit and how to use it for unit testing with Camel. You looked at using the mock component to easily write tests with expectations, run tests, and have Camel verify whether the expectations were satisfied. You also saw how to use the mock component to simulate a real component. You may wonder whether there's a more cunning way to simulate a real component than by using a mock, and there is. You'll look at simulating errors next, but the techniques involved could also be applied to simulating a real component.

9.4 Simulating errors

This section covers how to test that your code works when errors happen.

If your servers are on the premises and within a certain vicinity, you could test for errors by unplugging network cables and swinging an axe

at the servers, but that's a bit extreme. If your servers are hosted in the cloud, you'll have a long walk and have to go all Chuck Norris to enter the guarded data centers of Amazon, Google, or whoever your cloud providers are. That's sadly not going to happen in our lifetime; we can only dream about becoming Neo in *The Matrix*, capable of learning ninja and kung-fu skills in a matter of minutes.

Back to our earthly lives—you have to come up with something you can do from your computer. You'll look at how to simulate errors in unit tests by using the three techniques listed in table 9.7.

Table 9.7 Three techniques for simulating errors

Technique	Description
Processor	Using processors is easy, and they give you full control as a developer. This technique is covered in section 9.4.1.
Mock	Using mocks is a good overall solution. Mocks are fairly easy to apply, and they provide a wealth of other features for testing, as you saw in section 9.3. This technique is covered in section 9.4.2.
Interceptor	This is the most sophisticated technique because it allows you to use an existing route without modifying it. Interceptors aren't tied solely to testing; they can be used anywhere and anytime. This technique is covered in section 9.4.3.

9.4.1 Simulating errors using a processor

Errors are simulated in Camel by throwing exceptions, which is exactly how errors occur in real life. For example, Java will throw an exception if it can't connect to a remote server. Throwing such an exception is easy—you can do that from any Java code, such as from a `Processor`. That's the topic of this section.

To illustrate this, we'll take the use case from Rider Auto Parts: you're uploading reports to a remote server using HTTP, and you're using FTP as a fallback method. This allows you to simulate errors with HTTP connectivity.

The route from listing 11.14 is repeated here:

```
errorHandler(defaultErrorHandler()
  .maximumRedeliveries(5).redeliveryDelay(10000));

onException(IOException.class).maximumRedeliveries(3)
  .handled(true)
  .to("ftp://gear@ftp.rider.com?password=secret");

from("file:/rider/files/upload?delay=15m")
  .to("http://rider.com?user=gear&password=secret");
```

What you want to do now is simulate an error when sending a file to the HTTP service, and you'll expect that it'll be handled by `onException` and uploaded using FTP instead. This will ensure that the route is working correctly.

Because you want to concentrate the unit test on the error-handling aspect and not on the components used, you can mock the HTTP, FTP, and File endpoints. This frees you from the burden of setting up HTTP and FTP servers and leaves you with a simpler route for testing:

```
errorHandler(defaultErrorHandler()
  .maximumRedeliveries(5).redeliveryDelay(1000));

onException(IOException.class).maximumRedeliveries(3)
  .handled(true)
  .to("mock:ftp");

from("direct:file")
  .to("mock:http");
```

This route also reduces the redelivery delay from 10 seconds to 1 second, to speed up unit testing. Notice that the file endpoint is replaced with the

direct endpoint that allows you to start the test by sending a message to the direct endpoint; this is much easier than writing a file.

To simulate a communication error when trying to send the file to the HTTP endpoint, you add a processor to the route that forces an error by throwing a `ConnectException` exception:

```
from("direct:file")
  .process(new Processor()) {
    public void process(Exchange exchange) throws Exception {
      throw new ConnectException("Simulated error");
    }
  })
  .to("mock:http");
```

And with Java 8 DSL, it can be a bit shorter:

```
from("direct:file")
  .process(e -> { throw new ConnectException("Simulated error"); })
  .to("mock:http");
```

You then write a test method to simulate this connection error, as follows:

```
@Test
public void testSimulateConnectionError() throws Exception {
  getMockEndpoint("mock:http").expectedMessageCount(0);
  getMockEndpoint("mock:ftp").expectedBodiesReceived("Camel rocks");

  template.sendBody("direct:file", "Camel rocks");

  assertMockEndpointsIsSatisfied();
}
```

You expect no messages to arrive at the HTTP endpoint because you predicted the error would be handled and the message would be routed to the FTP endpoint instead.

The book's source code contains this example. You can try it by running the following Maven goal from the chapter9/error directory:

```
mvn test -Dtest=SimulateErrorUsingProcessorTest
mvn test -Dtest=SimulateErrorUsingProcessorJava8Test
```

Using the `Processor` is easy, but you have to alter the route to insert the `Processor`. When testing your routes, you might prefer to test them *as is* without changes that could introduce unnecessary risks. What if you could test the route without changing it at all? The next two techniques do this.

9.4.2 Simulating errors using mocks

You saw in section 9.3.5 that the mock component could be used to simulate a real component. But instead of simulating a real component, you can use what you learned there to simulate errors. If you use mocks, you don't need to alter the route; you write the code to simulate the error directly into the test method, instead of mixing it in with the route. The following listing shows this.

Listing 9.18 Simulating an error by throwing an exception from the mock endpoint

```
@Test
public void testSimulateConnectionErrorUsingMock() throws Exception {
  getMockEndpoint("mock:ftp").expectedMessageCount(1);

  MockEndpoint http = getMockEndpoint("mock:http");
  http.whenAnyExchangeReceived(new Processor() {
    public void process(Exchange exchange) throws Exception {
      throw new ConnectException("Simulated connection error");
    }
  });

  template.sendBody("direct:file", "Camel rocks");

  assertMockEndpointsSatisfied();
}
```

To simulate the connection error, you need to get hold of the HTTP mock endpoint, where you use the `whenAnyExchangeReceived` method to set a

custom `Processor`. That `Processor` can simulate the error by throwing the connection exception.

By using mocks, you put the code that simulates the error into the unit-test method, instead of in the route, as is required by the processor technique.

The source code contains an example in the chapter9/error directory that you can try by using the following Maven goal:

```
mvn test -Dtest=SimulateErrorUsingMockTest
```

Now let's look at the last technique for simulating errors.

9.4.3 Simulating errors using interceptors

Suppose your boss wants you to write integration tests for the example from section 9.4.1 that should, among other things, test what happens when communication with the remote HTTP server fails. How can you do that? It's tricky because you don't have control over the remote HTTP server, and you can't easily force communication errors in the network layer. Luckily, Camel provides features to address this problem. We'll get to that in a moment, but first you need to look at interceptors, which provide the means to simulate errors.

In a nutshell, an *interceptor* allows you to intercept any given message and act on it. Figure 9.5 illustrates where the interception takes place in a route.

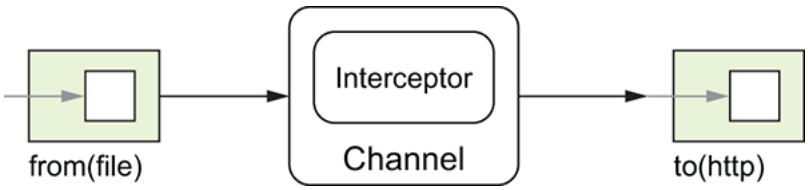

Figure 9.5 The channel acts as a controller, and it's where messages are intercepted during routing.

Figure 9.5 shows a low-level view of a Camel route, whereby you route messages from a file consumer to an HTTP producer. In between sits the channel, which acts as a controller, and this is where the interceptors (among others) live.

CHANNELS PLAY A KEY ROLE

Channels play a key role in the internal Camel routing engine, handling such things as routing the message to the next designated target, error handling, interception, tracing messages, and gathering metrics.

Table 9.8 lists three types of interceptors that Camel provides out of the box.

Table 9.8 The three flavors of interceptors provided out of the box in Camel

Interceptor	Description
`intercept`	Intercepts every single step a message takes. This interceptor is invoked continuously as the message is routed.
`interceptFromEndpoint`	Intercepts incoming messages arriving at a particular endpoint. This interceptor is invoked only once.
`interceptSendToEndpoint`	Intercepts messages that are about to be sent to a particular endpoint. This interceptor is invoked only once.

To write integration tests, you can use `interceptSendToEndpoint` to intercept messages sent to the remote HTTP server and redirect them to a processor that simulates the error, as shown here:

```
interceptSendToEndpoint("http://rider.com/rider")
  .skipSendToOriginalEndpoint()
  .process(new SimulateHttpErrorProcessor());
```

When a message is about to be sent to the HTTP endpoint, it's intercepted by Camel, and the message is routed to your custom processor, where you simulate an error. When this detour is complete, the message would normally be sent to the originally intended endpoint, but you instruct Camel to skip this step using the `skipSendToOriginalEndpoint` method.

TIP The last two interceptors in table 9.8 support using wildcards (`*`) and regular expressions in the endpoint URL. You can use these techniques to intercept multiple endpoints or to be lazy and just match all HTTP endpoints. We'll look at this in a moment.

Because you're doing an integration test, you want to keep the original route *untouched*, which means you can't add interceptors or mocks directly in the route. Because you still want to use interceptors in the route, you need another way to somehow add the interceptors. Camel provides the `adviceWith` method to address this.

9.4.4 Using adviceWith to add interceptors to an existing route

The `adviceWith` method, available during unit testing, allows you to add such things as interceptors and error handling to an existing route.

To see how this works, let's look at an example. The following code snippet shows how to use `adviceWith` in a unit-test method:

```
@Test
public void testSimulateErrorUsingInterceptors() throws Exception {
  RouteDefinition route = context.getRouteDefinitions().get(0);   ❶
```

❶ Selects the first route to be advised

```
  route.adviceWith(context, new RouteBuilder() {   ❷
```

❷ Uses adviceWith to add interceptor to route

```
    public void configure() throws Exception {
      interceptSendToEndpoint("http://*")
        .skipSendToOriginalEndpoint()
        .process(new SimulateHttpErrorProcessor());
```

```
        }
    });
    ..
}
```

The key issue when using `adviceWith` is to know which route to use. Because you have only one route in this case, you can refer to the first route enlisted in the route definitions list ❶. The route definitions list contains the definitions of all routes registered in the current `CamelContext`.

When you have the route, it's only a matter of using the `adviceWith` method ❷, which uses `RouteBuilder`. In the `configure` method, you can use the Java DSL to define the interceptors. Notice that the interceptor uses a wildcard to match all HTTP endpoints.

TIP If you have multiple routes, you'll need to select the correct route to be used. To help select the route, you can assign unique IDs to the routes, which you then can use to look up the route, such as `context.getRouteDefinition("myCoolRoute")`.

We've included this integration test in the book's source code, in the chapter9/error directory. You can try the test by using the following Maven goal:

```
mvn test -Dtest=SimulateErrorUsingInterceptorTest
```

TIP Interceptors aren't only for simulating errors; they're a general-purpose feature that can also be used for other types of testing. For example, when you're testing production routes, you can use interceptors to detour messages to mock endpoints.

This example only scratches the surface of `adviceWith`. Let's take a moment to cover what else you can do.

9.4.5 Using adviceWith to manipulate routes for testing

`adviceWith` is used for testing Camel routes that are *advised* before testing. Here are some of the things you can do by advising:

- Intercept sending to endpoints
- Replace incoming endpoint with another
- Take out or replace node(s) of the route
- Insert new node(s) into the route
- Mock endpoints

All these features are available from `AdviceWithRouteBuilder`, which is a specialized `RouteBuilder`. This builder provides the additional fluent builder methods listed in table 9.9.

Table 9.9 Additional `adviceWith` methods from AdviceWithRouteBuilder

Method	Description
`mockEndpoints`	Mocks all endpoints in the route.
`mockEndpoints(patterns...)`	Mocks all endpoints in the route that matches the pattern(s). You can use wildcards and regular expressions in the given pattern to match multiple endpoints.
`mockEndpointsAndSkip(patterns...)`	Mocks all endpoints and skips sending to the endpoint in the route that matches the pattern(s). You can use wildcards and regular expressions in the given pattern to match multiple endpoints.
`replaceFromWith(uri)`	Easily replaces the incoming endpoint in the route.
`weaveById(pattern)`	Manipulates the route at the node IDs that matches the pattern.
`weaveByToString(pattern)`	Manipulates the route at the node string representation (output from `toString` method) that matches the pattern.

Method	Description
`weaveByToUri(pattern)`	Manipulates the route at the nodes sending to endpoints matching the pattern.
`weaveByType(pattern)`	Manipulates the route at the node type that matches the pattern.
`weaveAddFirst`	Easily weaves in new nodes at the beginning of the route.
`weaveAddLast`	Easily weaves in new nodes at the end of the route.

The rest of this section covers the first four methods in table 9.9, and all the `weave` methods are covered in section 9.4.6.

Tell Camel that you're advising routes

Before you jump into the pool, you need to learn a rule from the life-guard. When `adviceWith` is being used, Camel will restart the advised routes. This occurs because `adviceWith` manipulates the route tree, and therefore Camel has to do the following:

- Stop the existing route
- Remove the existing route
- Add the route as a result of the `adviceWith`
- Start the new route

This happens for each of the advised routes during startup of your unit tests. It happens quickly: a route is started and then is immediately stopped and removed. This is an undesired behavior; you want to avoid restarts of the advised routes.

To solve this dilemma, you follow these three steps:

1. Tell Camel that the routes are being advised, which will prevent Camel from starting the routes during startup.
2. Advise the routes in your unit-test methods.
3. Start Camel after you've advised the routes.

The first step is done by overriding the `isUseAdviceWith` method from `CamelTestSupport` to return `true`:

```
public class MyAdviceWithTest extends CamelTestSupport {

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

Tells Camel that you intend to use adviceWith in this unit test

Then you advise the routes followed by starting Camel:

```
@Test
public void testMockEndpoints() throws Exception {
  RouteDefinition route = context.getRouteDefinition("quotes");
  route.adviceWith(context, new AdviceWithRouteBuilder() {
    public void configure() throws Exception {
      mockEndpoints();
    }
  });
  context.start();
  ...
}
```

(1) Advises the route

(2) Automocks all the endpoints

```
    }
  });

  context.start();    ❸
```

Starts Camel after the advice is done

```
  ...
```

To advise a route, you need to obtain the route by using its route ID. Then you can use the `AdviceWithRouteBuilder` ❶ that allows you to use the advice methods from table 9.9, such as automocking all the Camel endpoints. After you're finished with the advice, you need to start Camel, which would then start the routes after they've been advised, and thus no restart of the routes is performed anymore.

You can try this example, provided as part of the source code in the chapter9/advicewith directory, by using the following Maven goal:

```
mvn test -Dtest=AdviceWithMockEndpointsTest
```

What was the important message from the lifeguard? To tell Camel that you're using `adviceWith`.

Replace route endpoints

You may have built Camel routes that start from endpoints that consume from databases, message brokers, cloud systems, embedded devices, social streams, or other external systems. To make unit-testing these kind of routes easier, you can replace the route input endpoints with internal endpoints such as direct or SEDA endpoints. The following listing illustrates how to do this.

Listing 9.19 Replacing route endpoint from AWS SQS to SEDA for easy unit-testing

```
public class ReplaceFromTest extends CamelTestSupport {
```

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

❶ This test uses adviceWith.

```
    }

    @Test
    public void testReplaceFromWithEndpoints() throws Exception {
      RouteDefinition route = context.getRouteDefinition("quotes");    ❷
```

❷ Gets the route to be advised

```
      route.adviceWith(context, new AdviceWithRouteBuilder() {
        public void configure() throws Exception {
      replaceFromWith("direct:hitme");    ❸
```

❸ Replaces the route input endpoint with a direct endpoint

```
      mockEndpoints("seda:*");    ❹
```

❹ Mocks all SEDA endpoints

```
        }
      });

      context.start();    ❺
```

5

Starts Camel after you've finished advising

```
    getMockEndpoint("mock:seda:camel")    6
```

6

Sets expectations on the mock endpoints

```
            .expectedBodiesReceived("Camel rocks");    6
    getMockEndpoint("mock:seda:other")    6
            .expectedBodiesReceived("Bad donkey");    6

    template.sendBody("direct:hitme", "Camel rocks");    7
```

7

Sends in test messages to the direct endpoint

```
      template.sendBody("direct:hitme", "Bad donkey");

    assertMockEndpointsSatisfied();    8
```

8

Asserts the test passed by asserting the mocks

```
    }

    @Override
    protected RoutesBuilder createRouteBuilder() throws Exception {
      return new RouteBuilder() {
        public void configure() throws Exception {
    from("aws-sqs:quotes").routeId("quotes")    9
```

❾

The route originally consumes from Amazon SQS queue system.

```
            .choice()
              .when(simple("${body} contains 'Camel'")).to("seda:camel")
              .otherwise().to("seda:other");
        }
      };
    }
  }
```

The `ReplaceFromTest` class is using `adviceWith`, so you need to override the `isUseAdviceWithMethod` and return true ❶. Then you select the route to be advised ❷ by using `AdviceWithRouteBuilder`. The original input route is using the AWS-SQS component ❾, which you replace to use a direct component instead ❸. The route will send messages to other endpoints such as SEDA. To make testing easier, you automock all SEDA endpoints ❹. After the advice is done, you need to start Camel ❺. Because all the SEDA endpoints were automocked, you can use `mock:` as a prefix to obtain the mocked endpoints. This allows you to set expectations on those mock endpoints ❻. Then a couple of test messages are sent into the route by using the replaced endpoint, `direct:hitme` ❼. Finally, you check whether the test passes by asserting the mocks ❽.

You can try this example with the source code from the chapter9/advicewith directory using the following Maven goal:

```
mvn test -Dtest=ReplaceFromTest
```

Replacing the input endpoint in Camel routes by using `adviceWith` is just the beginning of the route manipulation capabilities available. The following section covers how to rip and tear your routes as you please.

9.4.6 Using weave with adviceWith to amend routes

When testing your Camel routes, you can use `adviceWith` to *weave* the routes before testing.

NAMING IN IT

Some IT professionals say that naming is hard, and they were right in terms of `adviceWith` and `weave`. The name *amend* might have been a better choice than *weave*, because what you're doing is amending the routes before testing. *Weaving* allows you to remove or replace parts of the routes that may not be relevant for a particular unit test that's testing other aspects of the routes. Another use case is to replace parts of routes that couple to systems that aren't available for testing or may otherwise slow testing.

In this section, you'll try some of the `weave` methods from `AdviceWithRouteBuilder` listed in table 9.9.

We'll start with a simple route that calls a bean that performs a message transformation:

```
from("seda:quotes").routeId("quotes")
  .bean("transformer").id("transform")
  .to("seda:lower");
```

Now suppose invoking the bean requires a connection to an external system that you don't want to perform in a unit test. You can use `weaveById` to select the bean and then replace it with another kind of transformation:

```
public void testWeaveById() throws Exception {
  RouteDefinition route = context.getRouteDefinition("quotes");
  route.adviceWith(context, new AdviceWithRouteBuilder() {
    public void configure() throws Exception {
  weaveById("transform").replace()    1
```

---

1 Uses weave to select the node to be replaced

```
      .transform().simple("${body.toUpperCase()}");    1

  weaveAddLast().to("mock:result");    2
```

❷ Routes to a mock endpoint at the end of the route

```
    }
  });
  context.start();    ❸
```

❸ Starts Camel after adviceWith

```
  getMockEndpoint("mock:result").expectedBodiesReceived("HELLO CAMEL");    ❹
```

❹ Expects the message to be in uppercase

```
    template.sendBody("seda:quotes", "Hello Camel");
    assertMockEndpointsSatisfied();
  }
```

In the original route, the bean call was given the ID `transform`, which is the ID you let `weaveById` ❶ use as select criteria. Then you select an *action* that can be one of the following:

- `remove` —Removes the selected nodes.
- `replace` —Replaces the selected nodes with the following.
- `before` —Before each of the selected nodes, the following is added.
- `after` —After each of the selected nodes, the following is added.

In this example, you want to replace the node, so you use `replace`, which is using the Java DSL to define a mini Camel route that's the replacement. In this example, you replace the `bean` with a `transform`.

You aren't restricted to only a 1:1 replacement, so you could, for example, provide before and after logging as shown here:

```
weaveById("transform").replace()
  .log("Before transformation")
  .transform().simple("${body.toUpperCase()}")
  .log("Transformation done");
```

To make testing easier, you use `weaveAddLast` ❷ to route to a mock endpoint at the end of the route. Remember to start Camel ❸ before you start sending messages into the route and assert that the test is correct ❹.

You can play with this example from the source code in chapter9/advicewith by running this Maven goal:

```
mvn test -Dtest=WeaveByIdTest
```

Using `weaveById` is recommended if you've assigned IDs to the EIPs in your Camel routes. If this isn't the case, what can you do then?

Weave without using IDs

When weaving a route, you need to use one of the `weaveBy` methods listed in table 9.9 as criteria to select one or more nodes in the route graph. Suppose you use the Splitter EIP in a route; then you can use `weaveByType` to select this EIP. We've prepared a little example for you that uses the Splitter EIP in the route shown here:

```
from("seda:quotes").routeId("quotes")
  .split(body(), flexible().accumulateInCollection(ArrayList.class))
    .transform(simple("${body.toLowerCase()}"))
    .to("mock:line")
  .end()
  .to("mock:combined");
```

TIP The aggregation strategy provided to the Splitter EIP uses a flexible fluent builder (highlighted in bold) that allows you to build common strategies. You can use this builder by statically importing the `org.apache.camel.util.toolbox.AggregationStrategies.flexible` method.

Because the route has only one Splitter EIP, you can use `weaveByType` to find this single splitter in the route. Using `weaveByType` requires you to pass in the model type of the EIP. The name of the model type uses the naming pattern **name**Definition, *so* the splitter is named `SplitDefinition`:

```
weaveByType(SplitDefinition.class)
  .before()
    .filter(body().contains("Donkey"))
    .transform(simple("${body},Mules cannot do this"));
```

Here you weave and select the Splitter EIP and then insert the following route snippet before the splitter. The route snippet is a message filter that uses a predicate to check whether the message body contains the word `"Donkey"`. If so, it performs a message transformation by appending a quote to the existing message body.

As usual, we've prepared this example of wisdom you can try from the chapter9/advicewith directory using the following Maven goal:

```
mvn test -Dtest=WeaveByTypeTest
```

Okay, this example has only one splitter. What if you have more than one splitter or want to weave a frequently used node such as `to`?

Selecting one or more by using weave

When using the `weaveBy` methods, they select all matching nodes, which can be anything from none, one, two, or more nodes. In those situations, you may want to narrow the selection to a specific node. This can be done by using the `select` methods:

- `selectFirst`—Selects only the first node.
- `selectLast`—Selects only the last node.
- `selectIndex(index)`—Selects only the *n*th node. The index is zero based.
- `selectRange(from, to)`—Selects the nodes within the given range. The index is zero based.

- `maxDeep(level)`—Limits the selection to at most *N* levels deep in the Camel route tree. The first level is number 1. So number 2 is the children of the first-level nodes.

Given the following route, you want to select the highlighted *to* node:

```
from("seda:quotes").routeId("quotes")
  .split(body(), flexible().accumulateInCollection(ArrayList.class))
    .transform(simple("${body.toLowerCase()}"))
    .to("seda:line")
  .end()
  .to("mock:combined");
```

You can then use `weaveByType` and `selectFirst` to match only the first found:

```
weaveByType(ToDefinition.class).selectFirst().replace().to("mock:line");
```

You can try this example from the accompanying source code in the chapter9/advicewith directory using the following Maven goal:

```
mvn test -Dtest=WeaveByTypeSelectFirstTest
```

If the route uses more *sent to* EIPs, the select criteria changes. You would then need to use `selectByIndex` and count the number of `to` occurrences from top to bottom, to find the index to use.

This last example can be made even easier by using `weaveByToUri`, which matches sending to endpoints:

```
weaveByToUri("seda:line").replace().to("mock:line");
```

You can try this yourself with the source code from the chapter9/advicewith directory by running the following Maven goal:

```
mvn test -Dtest=WeaveByToUriTest
```

SELECTING NODES BY USING PATTERNS

Both `weaveByToString` and `weaveByToUri` use pattern matching to match the nodes to select. The pattern algorithm is based on the same logic used by Camel interceptors. For example, to match all endpoints that send to a SEDA endpoint, you can use `"seda:*"`. Regular expressions can be used, such as `".*gold.*"`, to match any nodes that have the word *gold* in the endpoint URI.

That's all, folks, that we wanted to talk about `adviceWith`. It's a powerful feature in the right hands, so we recommend you give it a shot and play with the examples in the source code.

The two last sections of this chapter cover Camel integration testing. Section 9.5 focuses on integration testing using the Camel Test Kit. Section 9.6 goes out of town to look at using three third-party test frameworks with Camel.

Do you need a break? If so, now is a good time to put on the kettle for a cup of tea or coffee. The last two sections are best enjoyed having a cup of hot brew readily available.

9.5 Camel integration testing

So far in this chapter, you've learned that mocks play a central role when testing Camel applications. But integration testing often involves real live components, and substituting them with mocks isn't an option, because the point of the integration test is to test with live components. In this section, you'll look at how to test such situations without mocks.

Rider Auto Parts has a client application that business partners can use to submit orders. The client dispatches orders over JMS to an incoming order queue at the Rider Auto Parts message broker. A Camel application is then used to further process these incoming orders. Figure 9.6 illustrates this.

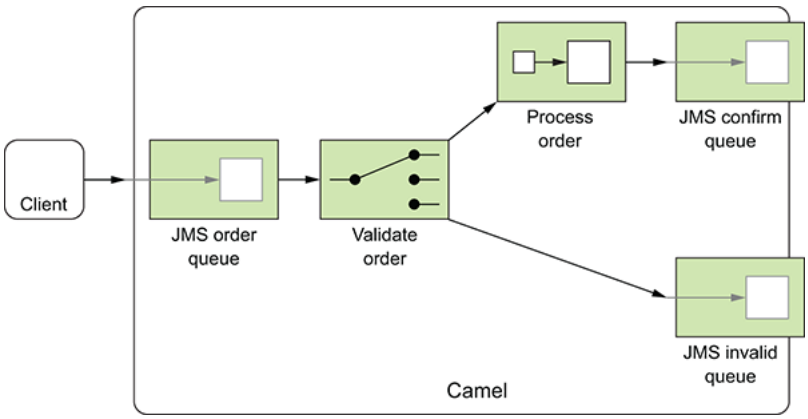

Figure 9.6 The client sends orders to an order queue, which is routed by a Camel application. The order is either accepted and routed to a confirm queue, or it's not accepted and is routed to an invalid queue.

The client application is written in Java, but it doesn't use Camel at all. The challenge you're facing is how to test that the client and the Camel application work as expected. How can you do integration testing?

9.5.1 Performing integration testing

Integration-testing the scenario outlined in figure 9.6 requires you to use live components. You must start the test by using the client to send a message to the order queue. Then you let the Camel application process the message. When this is complete, you'll have to inspect whether the message ended up in the right queue—the confirm or the invalid queue.

You have to perform these three tasks:

1. Use the client to send an order message.
2. Wait for the Camel application to process the message.
3. Inspect the confirm and invalid queues to see whether the message arrived as expected.

Let's tackle each step.

Use the client to send an order message

The client is easy to use. All you're required to do is provide an IP address to the remote Rider Auto Parts message broker and then use its `sendOrder` method to send the order.

The following code has been simplified in terms of the information required for order details:

```
OrderClient client = new OrderClient("localhost:61616");
client.sendOrder(123, date, "4444", "5555");
```

Wait for the Camel application to process the message

The client has sent an order to the order queue on the message broker. The Camel application will now react and process the message according to the route outlined in figure 9.6.

The problem you're facing now is that the client doesn't provide any API you can use to wait until the process is complete. You need an API that provides insight into the Camel application. All you need to know is when the message has been processed, and optionally whether it completed successfully or failed.

Camel provides `NotifyBuilder`, which provides such insight. Section 9.5.2 covers `NotifyBuilder` in more detail, but the following code shows how `NotifyBuilder` notifies you when Camel is finished processing the message:

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

OrderClient client = new OrderClient("tcp://localhost:61616");
client.sendOrder(123, date, "4444", "5555");

boolean matches = notify.matches(5, TimeUnit.SECONDS);
assertTrue(matches);
```

First, you configure `NotifyBuilder` to notify you when one message is done. Then you use the client to send the message. Invoking the `matches` method on the `notify` instance will cause the test to wait until the condition applies or until the five-second time-out occurs.

The last task tests whether `NotifyBuilder` matches. If it didn't match, it's because the condition was not met within the time-out period, and the assertion would fail by throwing an exception.

Inspect the queues to see if the message arrived as expected

After the message has been processed, you need to investigate whether the message arrived at the correct message queue. If you want to test that

a valid order arrived in the confirm queue, you can use the `BrowsableEndpoint` to browse the messages on the JMS queue. Using `BrowsableEndpoint`, you only peek inside the message queue, which means the messages will still be present on the queue.

Doing this requires a little bit of code, as shown here:

```
BrowsableEndpoint be = context.getEndpoint("activemq:queue:confirm",
                                           BrowsableEndpoint.class);
List<Exchange> list = be.getExchanges();
assertEquals(1, list.size());
String body = list.get(0).getIn().getBody(String.class);
assertEquals("OK,123,2017-04-20T15:47:58,4444,5555", body);
```

Using `BrowsableEndpoint`, you can retrieve the exchanges on the JMS queue using the `getExchanges` method. You can then use the exchanges to assert that the message arrived as expected.

The source code for the book contains this example in the chapter9/notify directory; you can try the example by using the following Maven goal:

```
mvn test -Dtest=OrderTest
```

We've now covered an example of performing integration testing without mocks. Along the road, we introduced `NotifyBuilder`, which has many more nifty features. We'll review those in the next section.

9.5.2 Using NotifyBuilder

`NotifyBuilder` is located in the `org.apache.camel.builder` package. It uses the Builder pattern, which means you stack methods on it to build an expression. You use it to define conditions for messages being routed in Camel. Then it offers methods to test whether the conditions have been met. We already used it in the previous section, but this time we'll raise the bar and show you how to build more complex conditions.

In the previous example, you used a simple condition:

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

This condition will match when one or more messages have been processed in the entire Camel application. This is a coarse-grained condition. Suppose you have multiple routes, and another message is processed as well. That would cause the condition to match even if the message you wanted to test is still in progress.

To remedy this, you can pinpoint the condition so it applies only to messages originating from a specific endpoint, as shown in bold:

```
NotifyBuilder notify = new NotifyBuilder(context)
  .from("activemq:queue:order").whenDone(1).create();
```

Now you've told the notifier that the condition applies only to messages that originate from the order queue.

Suppose you send multiple messages to the order queue, and you want to test whether a specific message was processed. You can do that using a predicate to indicate when the desired message was completed, using the `whenAnyDoneMatches` method, as shown here in bold:

```
NotifyBuilder notify = new NotifyBuilder(context)
  .from("activemq:queue:order")
  .whenAnyDoneMatches(
    body().isEqualTo("OK,123,2017-04-20'T'15:48:00,2222,3333")
  ).create();
```

In this example, you want the predicate to determine whether the body is equal to the expected result, which is the string starting with `OK,123,...`.

We've now covered some examples using `NotifyBuilder`, but the builder has many methods that allow you to build even more complex expressions. Table 9.10 lists the most commonly used methods.

Table 9.10 Noteworthy methods on NotifyBuilder

Method	Description
`from(uri)`	Specifies that the message must originate from the given endpoint. You can use wildcards and regular expressions in the given URI to match multiple endpoints. For example, you could use `from("activemq:queue:*")` to match any ActiveMQ queues.
`fromRoute(routeId)`	Specifies that the message must originate from the given route. You can use wildcards and regular expressions in the given `routeId` to match multiple routes.
`filter(predicate)`	Specifies the the message must match the given predicate.
`wereSentTo(uri)`	Matches when a message at any point during the route was sent to the endpoint with the given URI. You can use wildcards and regular expressions in the given URI to match multiple endpoints.
`whenDone(number)`	Matches when a minimum `number` of messages are done.
`whenCompleted(number)`	Matches when a minimum `number` of messages are

Method	Description
	completed.
`whenFailed (number)`	Matches when a minimum `number` of messages have failed.
`whenBodiesDone (bodies...)`	Matches when messages are done with the specified `bodies` in the given order.
`whenAnyDoneMatches (predicate)`	Matches when any message is done and matches the `predicate`.
`create`	Creates the notifier.
`matches`	Tests whether the notifier currently matches. This operation returns immediately.
`matches(timeout)`	Waits until the notifier matches or times out. Returns `true` if it matches, or `false` if a time-out occurs.

`NotifyBuilder` has more than 30 methods; we've listed only the most commonly used ones in table 9.10. Consult the online Camel documentation to see all the supported methods: http://camel.apache.org/notifybuilder.html.

NOTE `NotifyBuilder` works in principle by adding `EventNotifier` to the given `CamelContext`. `EventNotifier` then invokes callbacks during the routing of exchanges. This allows `NotifyBuilder` to listen for those events and react accordingly. `EventNotifier` is covered in chapter 16.

`NotifyBuilder` identifies three ways a message can complete:

- *Done*—The message is done, regardless of whether it completes or fails.
- *Completed*—The message completes with success (no failure).
- *Failed*—The message fails (for example, an exception is thrown and not handled).

The names of these three ways are also incorporated in the names of the builder methods: `whenDone`, `whenCompleted`, and `whenFailed` (listed in table 9.10).

TIP You can create multiple instances of `NotifyBuilder` if you want to be notified of different conditions. `NotifyBuilder` also supports using binary operations (`and`, `or`, `not`) to stack together multiple conditions.

The book's source code contains examples of using `NotifyBuilder` in the chapter9/notify directory. You can run them by using the following Maven goal:

```
mvn test -Dtest=NotifyTest
```

We encourage you to take a look at this source code and the online documentation.

So far, we've covered testing Camel by using its own set of test modules, a.k.a. the Camel Test Kit. But we don't live in a world of only Camels (although the authors of this book may have Camel too much on their minds). Numerous great test libraries are available for you to use. We'll end this chapter by presenting some of these test libraries you can use with Camel.

9.6 Using third-party testing libraries

Over the years, several great testing frameworks have emerged. They're worth introducing to you, and we provide basic examples of using them to test your Camel applications. These test frameworks aren't a direct replacement for the Camel Test Kit. These frameworks are suited for integration and system testing, whereas the Camel Test Kit is particularly strong for unit testing.

The third-party testing libraries covered in this section are as follows:

- *Arquillian*—Helps test applications that run inside a Java EE container
- *Pax Exam*—Helps test applications that run inside an OSGi container
- *Rest Assured*—Java DSL for easy testing of REST services
- *Other testing libraries*—A brief list of other kinds of testing libraries

The first two are test frameworks for system and integration testing. Rest Assured is a great test library that makes sending and verifying REST-based applications using JSON/XML payloads easier. You'll use this library in the first subsection to follow. At the end of the section, we quickly cover other testing libraries that we want to bring to your attention.

We start from the top with Arquillian.

9.6.1 Using Arquillian to test Camel applications

Arquillian was created to help test applications running inside a Java EE server. Readers who have built Java EE applications may know that testing your deployments and applications on said servers isn't an easy task. The bigger and older the Java EE server is, the worse that task is. It's not unusual to find yourself developing and testing locally, using a lighter Java EE server such as JBoss Application Server, and then crossing your fingers when testing on a production-like system.

The use of Java EE servers has peaked, and Arquillian has also evolved to become a testing framework that allows developers to create automated integration, functional, and acceptance tests. Arquillian can also be used for integration testing Docker containers and much more.

TIP You can learn a lot more about Arquillian from *Testing Java Microservices* by Alex Soto Bueno and Jason Porter (Manning, 2018).

In this section, we'll show you how to build an integration test with Arquillian that tests a Camel application running as a web application on a Java EE server such as Jetty or WildFly.

The Camel Example

The Camel application is a simple REST service that runs as a web application. The meat of this application is a Camel RESTful service that's defined using XML DSL notation in the quote.xml file, as shown in the following listing.

Listing 9.20 Camel RESTful service defined using REST DSL (quote.xml)

```
<bean id="quote" class="camelinaction.Quotes"/>     (1)
```

(1) A bean that returns a quote

```
<camelContext xmlns="http://camel.apache.org/schema/spring">

  <restConfiguration component="servlet"/>     (2)
```

(2) Uses the servlet component as the RESTful HTTP server

```
  <rest produces="application/json">     (3)
```

(3) RESTful service that produces JSON as output

```
    <get uri="/say">    ❹
```

❹

GET /say service

```
      <to uri="bean:quote"/>    ❺
```

❺

Calls the bean to get a quote to return as response

```
    </get>
  </rest>

</camelContext>
```

This code uses Camel's Rest DSL to define a RESTful service. This is a sneak peak of what's coming in the next chapter, where REST services are on the menu. The example uses a Java bean that provides a quote from a wise man ❶. The REST service uses the servlet component ❷ to tap into the servlet container of the Java EE server. The REST service has one `GET` service ❹ that calls the bean ❺ and returns the response as JSON ❸.

The quote bean is a simple class with a single method returning one of the three great wise sayings, as shown in the following listing.

Listing 9.21 A quote bean with great universal wisdom

```
public class Quotes {

  public static String[] QUOTES = new String[]{"Camel rocks",
                                    "Donkeys are bad", "We like beer"}
  public String say() {
    int idx = new Random().nextInt(3);
    return String.format("{\"quote\": \"%s\"}", QUOTES[idx]);
  }
}
```

The final piece of the application is the web.xml file, shown next.

Listing 9.22 web.xml to set up Camel servlet and bootstrap the Camel XML-based application

```
<web-app>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:camelinaction/quote.xml</param-value>    (1)
```

(1) Location of the Camel XML DSL file

```
  </context-param>

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener    (2)
```

(2) Uses Spring to bootstrap the Camel XML application

```
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>CamelServlet</servlet-name>
    <servlet-class>org.apache.camel.component.servlet.CamelHttpTransportServlet</
```

(3) Sets up the Camel Servlet

```
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
```

```
    <servlet-name>CamelServlet</servlet-name>
    <url-pattern>/*</url-pattern>    ❹
```

❹ Uses the root path for the Camel Servlet mapping

```
  </servlet-mapping>

</web-app>
```

This web.xml file is an old and trusty way of bootstrapping a web application. At first, you reference the classpath location of the Spring XML file ❶ that includes the Camel route. Then you use Spring listener ❷ to bootstrap your application. The Camel Servlet component is configured as a servlet ❸ with a URL mapping ❹.

We'll reuse this example or a modification thereof in the following sections, so we took a bit of time in this section to present the full code listing. Now it's time to pay attention, because Arquillian enters the stage.

Setting up Arquillian

When using Arquillian, you need to take three steps to create and use an Arquillian test:

1. Add Arquillian artifacts to Maven pom.xml.
2. Choose the application server to run the tests.
3. Write a test class with the deployment unit and the actual test.

Adding the Arquillian artifacts

When using Arquillian, it's recommended to import the Arquillian Maven BOM in your pom.xml file, which makes setting up the right Maven dependencies easier. This is done by adding `arquillian-bom` and `arquillian-junit-container` as dependencies to the Maven pom.xml file.

Choosing the application server

Because Arquillian runs the tests inside a Java EE or web container, you need to pick which one to use for testing. We used WildFly in section 9.2,

so let's use a different container this time—we'll choose Jetty by adding `arquillian-jetty-embedded-9` as a dependency to the Maven pom.xml file.

Adding the `arquillian-jetty-embedded-9` dependency doesn't include Jetty itself, so you need to add `jetty-webapp`, `jetty-deploy`, and `jetty-annotations` as well.

The Jetty dependencies should be set with `<scope>runtime</scope>` because Jetty is used only as the runtime Java EE container—you don't want to have compile-time dependencies on Jetty. You want your Camel web application to be portable and to run on other Java EE containers.

You can see the exact details of these Maven dependencies from the source code located in the chapter9/arquillian-web directory.

You're now ready to write the integration test.

Writing the test class with the deployment unit and actual test

Let's jump straight into code that shows the Arquillian test.

Listing 9.23 Arquillian test that creates a deployment unit and test method

```
@RunWith(Arquillian.class)    (1)
```

(1) Runs the test with Arquillian

```
public class QuoteArquillianTest {

  @Deployment    (2)
```

(2) Sets up the deployment unit in this method

```
  public static Archive<?> createTestArchive() {
    return ShrinkWrap.create(WebArchive.class)    ❸
```

❸ Uses ShrinkWrap to create a micro deployment unit

```
        .addClass(Quotes.class)    ❸
        .setWebXML(Paths.get("src/main/webapp/WEB-INF/web.xml").toFile());    ❸
    }

    @Test
    @RunAsClient    ❹
```

❹ Runs this test as a client

```
    public void testQuote(@ArquillianResource URL url) throws Exception {    ❺
```

❺ Injects the URL of the deployed application

```
        given().    ❻
```

❻ Uses Rest Assured to verify REST call

```
            baseUri(url.toString()).    ❻
        when().    ❻
            get("/say").    ❻
        then().    ❻
            body("quote", isIn(Quotes.QUOTES));    ❻
    }
}
```

An Arquillian test is a JUnit test that's been annotated with `@RunWith(Arquillian.class)` ❶. The deployment unit is defined by a static method that's been annotated with `@Deployment` ❷. The method creates an `Archive` using ShrinkWrap; you're in full control of what to include in the deployment unit ❸. In this example, you need only the `Quotes` class and the web.xml file.

THE BIG PICTURE OF SHRINKWRAP

ShrinkWrap is used to build the test archive that's deployed into the application server. This allows you to build isolated test deployments that include only what you need. From the application server point of view, the ShrinkWrap archive is a real deployment unit.

The Camel application you want to test uses the Camel servlet component. Therefore, you want to start the test from the client by calling the servlet, and you need to annotate the test with `@RunAsClient` ❹. Otherwise, the test will be run inside the application server. The URL to the deployment application is injected into the test method by the `@ArquillianResource` annotation ❺.

The implementation of the test method uses the Rest Assured test library that provides a great DSL to define the REST-based test ❻. The code should reason well with Camel riders who are using the Camel DSL to define integration routes.

This example is provided with the source code in the chapter9/arquillian-web directory; you can try the example using the following Maven goal:

```
mvn test
```

There's a lot more to Arquillian than what we can cover in this book. If you're running Camel applications in an application server, we recommend that you give Arquillian a try.

And speaking of application servers, let's take a look at a different kind: the OSGi-based Apache Karaf.

9.6.2 Using Pax Exam to test Camel applications

Pax Exam is for OSGi what Arquillian is for Java EE servers. If the mountain won't come to Muhammad, Muhammad must go to the mountain. This phrase covers the principle of both Arquillian and Pax Exam. Your tests must go to those application servers.

Earlier in this chapter, section 9.2.4 covered testing Camel by using OSGi Blueprint with the camel-test-blueprint module. This module has its limitations. In this section, you'll use Pax Exam to test the Camel quote application built in the previous section. Last time, the quote application was deployed as a web application in Jetty; this time, you'll use OSGi and deploy the application as an OSGi bundle.

Migrating the Camel application from web application to OSGi bundle

You migrate the Camel application from a web application by doing the following:

1. Change the pom.xml from a WAR file to an OSGi bundle.
2. Add the Maven Bundle plugin to the pom.xml file.
3. Create a features.xml file to make installing the application easy.
4. Convert the Spring XML file to OSGi Blueprint XML.

You start by changing the pom.xml from a web application to OSGi by setting `packaging` to `bundle`:

```
<packaging>bundle</packaging>
```

Then in the plugins section, you add the Maven Bundle Plugin that will generate the OSGi information in the MANIFEST.MF file, which turns the generated JAR into an OSGi bundle.

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>3.3.0</version>
  <extensions>true</extensions>
</plugin>
```

You then add another plugin that attaches the features.xml file to the project as a Maven artifact. This ensures that the features file gets installed and released to Maven, making it easy to install this example in Apache Karaf. The features.xml file defines which Camel and Karaf features this example requires and will be installed together with the example itself:

```
<features xmlns="http://karaf.apache.org/xmlns/features/v1.2.0"
          name="chapter9-pax-exam">
  <feature name="camel-quote" version="2.0.0">
    <feature>war</feature>    (1)
```

(1) Installs the Karaf WAR feature, which includes HTTP servlet support

```
    <feature>camel-blueprint</feature>    (2)
```

(2) Installs these Camel components

```
    <feature>camel-servlet</feature>    (2)
    <bundle>mvn:com.camelinaction/chapter9-pax-exam/2.0.0</bundle>    (3)
```

(3) Installs the Camel example itself

```
  </feature>
</features>
```

The biggest change in the example is the migration from Spring XML to OSGi Blueprint. It's not the differences between Spring and Blueprint that result in the big changes, as the syntax is similar. It's the fact that you need to set up the Camel servlet component to use the OSGi HTTP service in order to tap into the servlet container from the Apache Karaf server. The following listing shows how this is done.

Listing 9.24 Camel OSGi Blueprint using servlet with OSGi HTTP service

```
<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
             https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

<reference id="httpService"
      interface="org.osgi.service.http.HttpService"/>     (1)
```

(1) Refers to use the OSGi HTTP Service

```
<bean class="org.apache.camel.component.servlet.osgi.OsgiServletRegister
    init-method="register"     (2)
```

(2) Sets up a Servlet that uses /camel as context-path

```
      destroy-method="unregister">
  <property name="alias" value="/camel"/>
  <property name="httpService" ref="httpService"/>
  <property name="servlet" ref="quotesServlet"/>
</bean>

<bean id="quotesServlet"     (3)
```

(3) Camel servlet

```
      class="org.apache.camel.component.servlet.CamelHttpTransportServlet

<bean id="quote" class="camelinaction.Quotes"/>
```

Bean with the quotes

```
<camelContext id="quotesCamel"
                xmlns="http://camel.apache.org/schema/blueprint">

  <restConfiguration component="servlet"
```

Sets up a Servlet that uses /camel as context-path

```
                    port="8181" contextPath="/camel"/>

  <rest produces="application/json">
```

RESTful service that produces JSON as output

```
    <get uri="/say">
```

GET /say service

```
      <to uri="bean:quote"/>
```

Calls the bean to get a quote to return as response

```
    </get>
  </rest>
</camelContext>

</blueprint>
```

The Camel application uses the servlet container provided by Apache Karaf, by referring to the OSGi HTTP service ❶. Then you use a bit of XML

to set up a servlet that uses the context-path `/camel` ❷ and the Camel servlet to be used ❸. The rest of the example is similar to the example used when testing with Arquillian.

Running the example

This example is included with the source code in chapter9/pax-exam, and you can build and run the example in Apache Karaf. First, you need to build the example from Maven using the following:

```
mvn install
```

Then you can start Apache Karaf. From the Karaf shell, type the following commands:

```
feature:repo-add camel 2.20.1
feature:install camel
feature:repo-add mvn:com.camelinaction/chapter9-pax-exam/2.0.0/xml/featu
feature:install camel-quote
```

And from a web browser, you can open the following URL:

```
http://localhost:8181/camel/say
```

The web browser should show a quote as the response, such as the following:

```
{
  quote: "Camel rocks"
}
```

Okay, now it's time to listen up and cover your hair in your eyes (hint: a little tribute to Nirvana performing their MTV unplugged album over two decades ago).

Setting up Pax Exam

You need to take two steps to create and use a Pax Exam test:

1. Add Pax Exam artifacts to Maven pom.xml.
2. Write a test class with the deployment unit and the actual test.

Adding Pax Exam Artifacts

Adding Pax Exam to Maven pom.xml requires adding the following dependencies: `pax-exam`, `pax-exam-junit4`, `pax-exam-inject`, `pax-exam-link-mvn`, and `pax-exam-container-karaf`.

You can find the full list of the Pax Exam Maven artifacts in the source code for this example, located in the chapter9/pax-exam directory.

Writing the integration test class

Let's jump straight into the code and follow up with an explanation of the important details.

Listing 9.25 Pax Exam Integration test class with deployment and actual test

```
@RunWith(PaxExam.class)    (1)
```

(1) Runs this test with Pax Exam

```
public class PaxExamIT {

@Inject
@Filter("(camel.context.name=quotesCamel)")    (2)
```

(2) Injects the CamelContext from the application

```
protected CamelContext camelContext;

@Configuration    (3)
```

(3) Sets up the deployment unit

```
public Option[] config() {
  return new Option[]{
  karafDistributionConfiguration()  4
```

4

Configures the Apache Karaf server to use

```
    .frameworkUrl(maven().groupId("org.apache.karaf")  4
    .artifactId("apache-karaf").version("4.1.2").type("tar.gz"))  4
    .karafVersion("4.1.2")  4
    .name("Apache Karaf")  4
    .useDeployFolder(false)  4
    .unpackDirectory(new File("target/karaf")),  4

  keepRuntimeFolder(),  5
```

5

Keeps the runtime folder so we can look there in case of errors

```
    configureConsole().ignoreRemoteShell(),
  logLevel(LogLevelOption.LogLevel.WARN),  6
```

6

Sets the logging level to not be verbose

```
    junitBundles(),
  features(getCamelKarafFeatureUrl(), "camel", "camel-test"),  7
```

7

Installs Apache Camel

```
  features(getCamelKarafFeatureUrl(), "camel-http4"),  8
```

8

Installs the camel-http4 component we use in this test

```
    features(maven().groupId("com.camelinaction")    9
```

9

Installs this example using the Maven coordinates

```
        .artifactId("chapter9-paxexam").version("2.0.0")
        .classifier("features").type("xml"), "camel-quote")
    };
  }

  public static UrlReference getCamelKarafFeatureUrl() {
    return mavenBundle().
      groupId("org.apache.camel.karaf").artifactId("apache-camel")
        .version("2.20.1").classifier("features").type("xml");
  }

  @Test
  public void testPaxExam() throws Exception {    10
```

10

The actual test method that runs inside Karaf container

```
    long total = camelContext.getManagedCamelContext().getExchangesTotal()
    assertEquals("Should be 0 exchanges completed", 0, total);

    Thread.sleep(2000);    11
```

11

Pax-Exam needs a little delay before really ready

```
    String url = "http4://localhost:8181/camel/say";
```

```
    ProducerTemplate template = camelContext.createProducerTemplate();
  String json = template.requestBody(url, null, String.class);    ⓬
```

⓬

Calls the Camel Servlet and prints the response

```
    System.out.println("Wiseman says: " + json);

    total = camelContext.getManagedCamelContext().getExchangesTotal();
    assertEquals("Should be 1 exchanges completed", 1, total);
  }
```

As you can see, you create a Pax Exam integration test by annotating the test class with `@RunWith(PaxExam.class)` ❶. You can perform dependency injection by using `@Inject`, which injects the `CamelContext` of the Camel application being tested ❷. Then you configure what to deploy and run on the Apache Karaf server in the `@Configuration` method ❸. At first, you configure the Apache Karaf server ❹, which will be unpacked in the target/karaf directory and started from there. It's a good idea to keep the unpacked directory after the test ❺, which allows you to peek inside the Karaf server files, such as the log files. By default, Pax Exam is verbose in the logging, so you turn it up all the way to `WARN` level ❻ to keep the logging to a minimum. But you can adjust this to `INFO` or `DEBUG` to have a lot more logging in case something is wrong. Then you install Apache Camel ❼ and the camel-http4 component ❽, which you use for testing your Camel application that's installed next ❾.

The remainder of the code is the test method ❿, which uses the injected `CamelContext` ❷ to create a `ProducerTemplate` that you use to call the Camel servlet ⓬, after a little delay of two seconds ⓫. The delay is needed because Pax Exam starts the test too quickly, while Karaf isn't yet ready to accept traffic on its HTTP server. Notice you assert that Camel has processed no message before the test, and then one message afterward. The source code example includes an additional test to verify that the returned message is one of the expected quotes. But we left that code out of the listing.

Running the test

Now is a good time to run the Pax Exam integration test yourself from the chapter9/pax-exam directory, which is done via the following:

```
mvn integration-test -P pax
```

First-time users with Pax Exam will have a lot to learn and master, as setting up the `@Configuration` method can be tricky. Pax Exam has many options—more than what we can cover in this book. But we've given you a good start.

The last, brief section highlights numerous testing libraries. Each is phenomenal in its own way.

9.6.3 Using other testing libraries

Testing has many facets, and in this book we've chosen to focus on unit and integration testing with Camel; these disciplines have the strongest coupling to Camel. Other disciplines, such as load and performance testing, are more generally applicable. You can find other resources that cover those concepts well. We do have a list of noteworthy testing libraries we want to bring to your attention.

Awaitility

Testing asynchronous systems is hard. Not only does it require handling threads, time-outs, and concurrency issues, but the intent of the test code can be obscured by all these details. Awaitility (https://github.com/awaitility/awaitility) is a DSL that allows you to express expectations of an asynchronous system in a concise and easy-to-read manner.

Byteman

Byteman (http://byteman.jboss.org) is a bytecode manipulation tool that can inject custom code into specific points. For example, you can use Byteman to inject code that triggers errors to happen at certain places.

Citrus Integration Testing

Citrus (www.citrusframework.org) is a test framework that's intended to help you become successful with integration testing. With Citrus, you can build automated integration testing and have support for testing with Apache Camel. You can find a Camel and Citrus example with the source code in the chapter9/citrus directory.

Java Mission Control

Java Mission Control (JMC) is a part of Java that you can run from a shell. JMC is able to collect low-level details about running Java applications with little overhead (1% or less). With this tool, you can visually analyze the collected data to help you understand how your application is behaving on the JVM.

JMH

JMH (http://openjdk.java.net/projects/code-tools/jmh) is a Java harness for building, running, and analyzing micro benchmarks written in Java. With JMH, you set up a new Java project that can be created using a Maven archetype and you embed your application by adding the needed JARs/dependencies to the project. You then write benchmarks by writing small tests annotated with `@Benchmark`. You can easily specify to run tests in intervals of tens or thousands and also allow the JVM to warm up first.

Gatling

Gatling (http://gatling.io) is a load- and performance-testing library that provides a DSL enabling you to write load testing at a high abstraction level. Gatling provides beautiful and informative reports as output of running the tests.

YourKit

YourKit (https://yourkit.com) is a Java CPU and memory profiler. With YourKit, you can sample running JVMs and gather statistics and reports. YourKit comes with a graphical application for visualizing hot spots in your application, such as the most-used methods, or methods that use the most CPU time or memory. The Camel team has used this tool with success to find performance bottlenecks in the Camel routing engine.

WIRESHARK

Wireshark (www.wireshark.org) is a network packet analyzer. It's used for network troubleshooting and inspection of network protocols. Wireshark comes with a graphical interface that lets you see what happens at the lower networking level.

That's all, folks. We've reached the end of this topic.

9.7 Summary and best practices

Testing is a challenge for every project. It's generally considered bad practice to perform testing only at the end of a project, so testing often begins when development starts and continues through the remainder of the project lifecycle.

Testing isn't something you do only once, either. It's involved in most phases in a project. You should do unit testing while you develop the application. And you should implement integration testing to ensure that the various components and systems work together. You also have the challenge of ensuring that you have the right environments for testing.

Camel can't eliminate these challenges, but it does provide a great Camel Test Kit that makes writing tests with Camel applications easier and less time-consuming. You can run Camel in any kind of Java environment your like. We covered how to write Camel unit tests for the most popular ways of using Camel, whether it's running plain Camel standalone, with Spring in any form or shape, with CDI, OSGi, or on popular Java EE servers such as WildFly.

We also reviewed how to simulate real components using mocks in the earlier phases of a project, allowing you to test against systems you may not currently have access to. In chapter 11, you'll learn all about error handling; in this chapter you saw how to use the Camel Test Kit to test error handling by simulating errors.

We reviewed techniques for integration testing that don't involve using mocks. Doing integration testing, using live components and systems, is often harder than unit testing, in which mocks are a real advantage. In integration testing, mocks aren't available to use, so you have to use other

techniques such as setting up a notification scheme that can notify you when certain messages have been processed. This allows you to inspect the various systems to see whether the messages were processed as expected (such as by peeking into a JMS queue or looking at a database table).

Toward the end of this chapter, we covered testing Camel with numerous integration testing libraries that make it possible to write tests that deploy and run in the application server of choice. You saw how Arquillian makes it easier to run tests inside a Java EE server such as WildFly. For OSGi users, we covered how to use Pax Exam for running tests inside Apache Karaf containers.

Here are a few best practices to take away from the chapter:

- *Use unit tests*—Use the Camel Test Kit from the beginning and write unit tests in your projects.
- *Use the mock component*—The mock component is a powerful component for unit testing. Use it rigorously in your unit tests.
- *Amend routes before testing*—Learn how to use the advice feature that allows you to manipulate your Camel routes before running your unit tests.
- *Test error handling*—Integration is difficult, and unexpected errors can occur. Use the techniques you've learned in this chapter to simulate errors and test that your application is capable of dealing with these failures.
- *Use integration tests*—Build and use integration tests to test that your application works when integrated with real and live systems.
- *Run tests on an application server*—If you run your Camel applications on an application server, we recommend using Arquillian or Pax Exam to run tests that deploy and run on the application server.
- *Test REST services*—The third-party testing library Rest Assured is a great tiny library for writing human-understandable unit tests that call REST services. Like Camel, it provides a DSL that almost speaks in human terms about what's happening. Make sure to pick up this library if you jump on the hype of REST and JSON.
- *Automate tests*—Strive for as much test automation as possible. Running your unit tests as well as integration tests should be as easy

and repeatable as possible. Set up a continuous integration platform that performs testing on a regular basis and provides an instant feedback loop to your development team.

Speaking of REST, what a great segue to the next chapter, which is also a long chapter covering modern RESTful web services.