SAMPLE CHAPTER

MANNING

Spring Integration IN ACTION

Mark Fisher
Jonas Partner
Marius Bogoevici
Iwein Fuld

FOREWORD BY Rod Johnson

Spring Integration in Action

by Mark Fisher, Jonas Partner,
Marius Bogoevici, Iwein Fuld

Chapter 18

brief contents

18 Testing

This chapter covers

- Test-driven development in the context of messaging
- Hamcrest and Mockito matchers for messages
- Testing asynchronous applications

One of the great accomplishments of our industry over the last 20 years is *test-driven development (TDD)*. Where many methodologies have proven only to work in theory or have never proven their need, TDD has flourished. The reason for this is simple: clients only pay willingly for working software, and there's only one way to prove that software works—test it. In essence, TDD makes the developer responsible for proving that the software works. A green test is the ultimate proof of correctness. If you don't have a clue how you're going to test the application, you don't have any business building it.

There are many ways to test software. One of the oldest ways is to test it manually. Manual testing is still valid and in wide use because of its simplicity, but experienced developers dread the tedious work of so-called monkey testing. A lot of this work can be automated. Even better, if you can isolate a part of the program in such a way that it doesn't require every test fixture to simulate human interaction, writing tests becomes a simple development task with excellent return on investment. SUnit, invented by some of the bright minds that also started the Agile

movement, was ported to Java in the late 1990s. If you don't know JUnit yet, firmly pull the handbrake toward you and make sure you learn JUnit before you read any further.

After reading the rest of the book and being exposed to test code in the samples as well, you'll find little new here in terms of *what* you can do. The main thrust of this chapter is about *why* you should pick a certain option.

Because the topic of this chapter cuts across the other topics, this chapter is organized differently. Like the other chapters, it uses code samples from the sample application, but it doesn't focus on a particular use case. It also doesn't have a dedicated "Under the hood" section, first because the test framework code is simple enough to embed with its usage and also because the test code serves as an example of how to extend JUnit to deal with the messaging domain.

This chapter builds on top of JUnit tests from the sample to show you the intricate details of testing asynchronous and concurrent programs that were built using the pipes-and-filters architecture supported by Spring Integration. As you might've experienced, testing these types of applications is more convoluted than testing classical applications. When you're using Spring Integration, you'll find that it's often necessary to write tests that assert things about the payload of a message that's received from a channel or to write assertions about particular headers on such a message. The boilerplate code normally needed to do this is in large part taken care of by Spring Integration's own test framework. This test framework builds on top of Hamcrest to offer custom matchers that can be used with `assertThat`. It also has some convenience classes related to the asynchronous nature of certain Spring Integration components, such as `QueueChannel`.

In addition to Hamcrest matchers, Spring Integration tests can make use of Mockito extensions. Mocking services out of tests relating to the message flow through the system becomes more important as a system becomes more complex or when service implementations are developed on a different schedule than the configuration that makes up the message bus. This chapter discusses different use cases for the test module. The authors strongly believe that tests shouldn't hide complexity, so, as mentioned, we include no "Under the hood" section. Instead, you'll find all the details right with the usage examples.

Assertions with Hamcrest matchers

Since version 4.4, JUnit added Hamcrest support and in later versions also repackaged the Hamcrest framework. *Hamcrest* is a matching framework that allows a user to compare an object against a predefined matcher with a readable API. Hamcrest has a more generic use than just JUnit testing, but it's best known for its use in JUnit's `assertThat` method:

```
assertThat("Tango", is(not("Foxtrot")));
```

This makes both the test code and the thrown exceptions more readable.

> **Testing behavior with mocks**
>
> Mocking is used to allow assertions on behavior instead of state. Frameworks like EasyMock, JMock, and Mockito help create mocks that allow verification of behavior. Mockito is probably the simplest mocking framework around. Certain advanced features are not supported in Mockito, but that makes it an excellent candidate to use for illustration. If you're unfamiliar with mocking, you're encouraged to read "Mocks Aren't Stubs" by Martin Fowler (available at http://mng.bz/mq95).

To make use of all the test goodness, you should depend on `spring-integration-test` or `org.springframework.integration.test`, depending on whether you're using OSGi. This JAR is packaged separately from the main Spring Integration distribution, because we don't want to force transitive dependencies on Mockito and Hamcrest on all Spring Integration users:

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

You've just added another Spring Integration JAR on your classpath—now what? Let's look at what's inside that JAR.

18.1 Matching messages with the Spring Integration testing framework

What goes in must come out. When a message moves into the system, it must come out in some form or another, either through being consumed by an outbound channel adapter, as another message being sent on a subsequent channel, or as an `ErrorMessage` being sent to the `errorChannel`. In a test fixture, you're usually interested in the properties of the outgoing messages, but these properties might be hard to reach:

```
@Test
public void outputShouldContainDelayedFlight() {
    inputChannel.send(testMessage());
    Message output = outputChannel.receive();
    assertThat(((FlightDelayedEvent) output
                    .getPayload()).getDelay(),
                is(expectedDelay) );
}
```

As you can see here, getting to the delay requires a cast and two method invocations. Let's see if we can do better than that. In the next section, you'll see how to factor the unwrapping logic out of your test cases.

18.1.1 Unwrapping payloads

With Spring Integration's test module, you can use the matchers that deal with unwrapping internally. First we look at an example, then we look at the underlying details. Starting with the previous example, you probably already noticed some pain points in the test code. The main problem is in the assertion. The `is` matcher isn't particularly well suited to deal with messages.

Ideally, you'd have a matcher that can be used like this:

```
assertThat(outputChannel.receive(), hasPayload(expectedDelay));
```

It's no coincidence that with the `PayloadMatcher` you can do exactly this. All you need to do is add the following import statement:

```
import static org.springframework.integration.matcher.PayloadMatcher.*;
```

This gives you two methods related to payloads: `hasPayload(T payload)` and its overloaded cousin accepting `Matcher<T>`. This way, you can also use other variants of the theme:

```
assertThat(outputChannel.receive(), hasPayload(expectedDelay));
assertThat(outputChannel.receive(), hasPayload(same(expectedDelay)));
assertThat(outputChannel.receive(), hasPayload(is(FlightDelay.class)));
```

This makes your life as a Spring Integration user a lot easier, and the code you need is almost trivial. Let's look at the code of the `PayloadMatcher` in the following listing.

Listing 18.1 The `PayloadMatcher`

```
public class PayloadMatcher extends TypeSafeMatcher<Message<?>> {

    private final Matcher<?> matcher;

    PayloadMatcher(Matcher<?> matcher) {
        super(); this.matcher = matcher;
    }

    public boolean matchesSafely(Message<?> message) {
        return matcher.matches(message.getPayload());
    }

    public void describeTo(Description description) {
        description.appendText("a Message with payload: ")
          .appendDescriptionOf(matcher);
    }

    @Factory
    public static <T> Matcher<Message<?>> hasPayload(T payload) {
        return new PayloadMatcher(IsEqual.equalTo(payload));
    }

    @Factory
    public static <T> Matcher<Message<?>> hasPayload
    ➥(Matcher<T> payloadMatcher) {
        return new PayloadMatcher(payloadMatcher);
    }
}
```

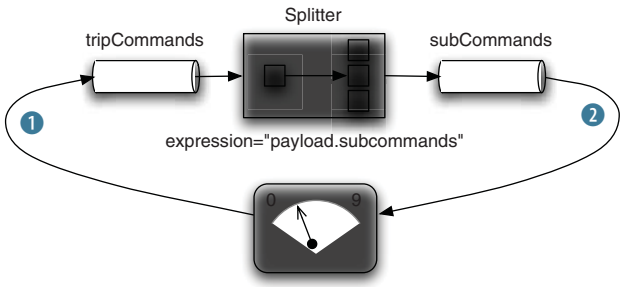

Figure 18.1 The test sends a message ❶ on the `tripCommands` channel and receives the subcommands that were sent by the splitter. Now a test can verify that the splitter is configured correctly by asserting that the payload ❷ of the messages matches the contents of the original `TripCommand`.

As you can see, this listing extends `TypeSafeMatcher` and implements two factories for the matcher. If your only concern is to match payloads, you might even opt to add this class to your project and avoid the extra dependency on the `spring-integration-test` JAR. A few more features are bundled in Spring Integration's test module. For example, you might require matching on headers too, as you'll see in the next section.

Matching messages is particularly useful if the framework is doing work that's important for business concerns. In many cases, the message payload is determined by business logic in Java code, and asserting things about the payload doesn't make much sense in an integration test (because most of that would already be covered in a unit test around the service). In some cases, though, such as when you use the Spring expression language, things change. Let's take another example from the sample application.

In the sample application, a user can fill out a form creating a new trip, and from that a `CreateTripCommand` is sent into the system wrapped in a message. The message goes through a splitter that chops the command into subcommands for rental cars, hotel rooms, and flights. Let's zoom in on the tests for the splitter. In figure 18.1 you can see how we modified the fixture to allow you to control incoming and outgoing messages.

You can now make sure the expression is correct in a controlled test case. All you need to do is receive three messages from the `javaLegQuoteCommands` channel and assert that their payloads meet your expectations.

The test case remains relatively simple as you can see in the following code:

```
@Autowired
MessageChannel tripCommands;

@Autowired
PollableChannel javaLegQuoteCommands;

@Test
public void splitterShouldSplitIntoSubcommands() {
    CreateTripCommand tripCommand = mock(CreateTripCommand.class);
    Message<CreateTripCommand> tripCommandMessage =
           MessageBuilder.withPayload(tripCommand).build();
    final Command carCommand = mock(Command.class);
    final Command flightCommand = mock(Command.class);
    final Command hotelCommand = mock(Command.class);
```

```
    given(tripCommand.getSubCommands()).willReturn(
             Arrays.asList(carCommand, flightCommand, hotelCommand));

    tripCommands.send(tripCommandMessage);
    List<Message<? extends Object>> received =
            Arrays.asList(javaLegQuoteCommands.receive(100),
               javaLegQuoteCommands.receive(100),
               javaLegQuoteCommands.receive(100));
    assertThat(received.size(), is(3));
}
```

The trick here is to plug into the existing system without replacing logic that you want to test. In this case, the `javaLegQuoteCommands` channel is overridden by a queue channel, and no other components are receiving from it.

As you saw in the previous example, it's simple and useful to write tests that make assertions on the payload of a message. More often than not, though, the headers of messages play at least as big a role in the integration of the system. In the next section, we go into the details of header matching.

18.1.2 Expectations on headers

In many cases, when you're testing the integrated application, it's more important to make assertions about the infrastructural effects on messages than on the business services' effects on messages. Typically, the effects of business services are already covered by unit tests, so you don't need to cover all the corner cases in your integration test again. But headers on messages are typically set by components that are decoupled from services and can only do their work in an integrated context. For headers set in this manner, you need to test all the corner cases in an integration test.

Let's look at the booking of a flight again. From the UI, a command describing the desired booking is submitted. This command is consumed by the booking service, which puts an event on the bus that signals the result of the booking (success or failure). To guarantee idempotence, the service activator for the booking service is preceded by a header enricher that stores a reference to the original command in the headers and a filter that drops any message containing a command that has already been executed. It's followed by a service activator that keeps track of all the successfully executed commands, for example, in a table that's also used by the filter.

This construction contains enough complexity and business value to make it the target of a test, but testing all these components in isolation doesn't assert anything about what Spring Integration will do with the header values. You need to make assertions about headers, which you could do manually:

```
@Test
public void outputHasOriginalCommandHeader() {
    //when
    inputChannel.send(testMessage());
    Message output = outputChannel.receive();
    //verify
    assertThat(
       (BookFlightCommand) output.getHeaders().get("command")
```

```
        , is(expectedCommand)
    );
}
```

But similar to matching payloads, matching headers manually causes smelly code. Again, there are matching facilities in Spring Integration's test module that can help you. The implementation is similar to the `PayloadMatcher`, so you only need to look at the usage here:

```
BookFlightCommand testBookFlightCommand =
        new BookFlightCommand("SFO", "ORD");
Message<?> testMessage =
        MessageBuilder.withPayload(testBookFlightCommand)
                .setCorrelationId("ABC")
                .build();
// send to flow where header-enricher stores payload as 'command'
inputChannel.send(testMessage);
Message<?> reply = outputChannel.receive();
assertThat(reply, hasHeaderKey("command"));
assertThat(reply, hasHeader("command", notNullValue()));
assertThat(reply, hasHeader("command", is(BookFlightCommand.class)));
assertThat(reply, hasHeader("command", testBookFlightCommand));
assertThat(reply, hasCorrelationId("ABC"));
// create a map of headers to be verified
Map<String, Object> map = new HashMap<String, Object>();
map.put("command", testBookFlightCommand);
map.put(MessageHeaders.CORRELATION_ID, "ABC");
assertThat(reply, hasAllHeaders(map));
```

As you can see, the header matching methods are very convenient and drastically reduce the amount of noise in test code. The example above demonstrates several of the matching options: checking for the presence of a header key, verifying that a header value is not null, validating a header value's type, and asserting that a header contains an expected instance. Moreover, all of the predefined header keys can be matched via explicitly named methods as shown above with the `hasCorrelationId` method. On the last line, you see that there's even a method for testing that all key-value pairs in a given map are present as headers on the message against which you match. That's far more convenient than iterating through the map directly and testing each key and value against the message headers. Not only is the test code more readable, but the error message produced by a failed assertion will provide much more detail than if you were testing individual values directly. If we change the correlation ID in the test message, for example, the resulting test failure message would contain the following:

```
  Expected: a Message with Headers containing an entry with key
      "correlationId" and value matching "ABC"
      got: <[Payload=CONFIRMATION-ID:123][Headers={correlationId=XYZ,...
```

This section established a solid foundation in terms of matching messages based on payloads and headers. Regarding the matchers, it isn't trivial to deal with the fact that a message received from a channel doesn't have the benefit of generics in many cases.

If you choose to implement your own matchers, you should expect to invest some of your time in fine-tuning parameterization.

Matching the message state is only part of the equation. You should also verify that service activators, transformers, and channel adapters invoke services correctly. For this, you can use mocks. When you're using a mocking framework, things get more complicated because you must deal with the particulars of the mocking framework as well. We outline the support for Mockito in the next section.

18.2 Mocking services out of integration tests

When your test subject has dependencies that aren't relevant for your test, you can use mocks or stubs to factor their influence out of the test fixture. People often refer to such refactoring as *mocking out*. A briefing on mocking is beyond the scope of this chapter, but we keep a strict definition of a mock as something that can be used to test behavior (and is usually created by a mocking framework), as opposed to a stub, which is typically used to test state and is created as an inner class in a test case.

In this chapter, we show only mocks using Mockito, which serves our need for concise and readable code samples. Other mocking frameworks or stubs can be used in the same manner; the particulars of Mockito are irrelevant to the point being made.

Most unit tests require a test harness that simulates the external dependencies (for example, through mocking or stubbing). But if configuration becomes a major part of the behavior of your application, as with Spring Integration, it becomes important to test the configuration itself. This means that it becomes sensible to mock out business code and let a message flow through the system just to see if it's handled correctly by the infrastructure. This would concern routing, filtering, header enrichment, and interaction with other systems. For example, you might want to pick up a file from a certain directory, set its name as a header value, then unzip the file and unmarshal it to domain objects. All this can be considered infrastructure—*customized infrastructure*, if you will.

Customized infrastructure is usually important to the business without being tightly related to a particular business use case. For example, properly setting a header is essential for your system to perform its tasks as designed, but setting this header is only a small part of the story. The particular header enricher has a place as a unit in the system, so it should have a designated unit test. If you're using SpEL, you have only XML configuration to test.

Let's say you have a header enricher that sets the original command as a header so it can be used later in the chain when the payload is already referencing the response:

```
<header-enricher>
    <header name="originatingCommand" expression="payload"/>
</header-enricher>
```

Even though the expression is trivially simple, this needs to be tested thoroughly. For example, a change that postpones the unmarshalling to a `BookingCommand` could cause a regression where the `originatingCommand` header suddenly references a `Document` instead.

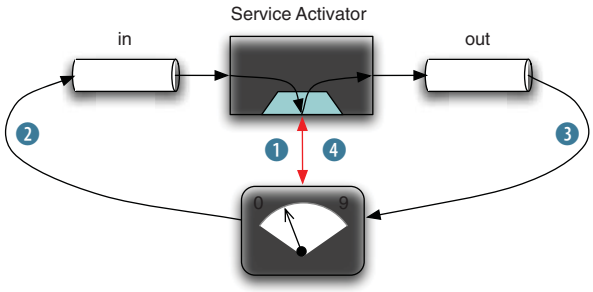

Figure 18.2 First record the behavior of the mock ❶. Then send a test message to the input channel ❷. After waiting for the message to come out of the other end ❸, verify that the appropriate operations on the mock have been invoked ❹. Variations on this recipe work also in more complex cases.

In figure 18.2 you can see how to generically set up a test that verifies the framework behavior. The example chosen is a service activator that invokes a method on a mock. Arguably, this setup would make sense only as an integration test for the framework, but it serves as a simple example. As the complexity of your configuration increases, it becomes increasingly useful to verify the flow of the messages through the system.

In a test like this, you're not interested in the behavior and effects of the service that receives the objects as message payloads. In fact, a failure in that service might distract you from the purpose of the test. It therefore makes sense to replace the service with a mock, but because this service is wired as a bean in a Spring context, it isn't as easy as injecting a component with mocked collaborators, as you would do in a normal unit test. But there's a trick you can use:

```
<bean id="service"
      factory-method="mock"
      class="org.mockito.Mockito">
   <constructor-arg value="example.ServiceToMock"/>
</bean>
```

This code overrides the `service` bean with a bean that's a mock created by Mockito. This bean, `@Autowired` into your test case, can be used like any other mock with the only difference being that its lifecycle will be managed by Spring instead of JUnit directly.

This strategy is particularly useful to avoid calling services that operate on external systems. Invoking an external system is more problematic to clean up, but it's also more complicated to verify the invocation happened correctly. If you use a mock, you can simply verify that it was touched, and that's it. This is a good option for channel adapters too, because it gives you a generic way to deal with them. In section 18.3, we use this strategy as well to deal with the need to wait for an invocation to happen before we start asserting the result.

This section focused only on Mockito, but similar support for EasyMock, JMock, and RMock can be implemented along the same lines. It's unlikely that Spring Integration will natively support all of these frameworks in the near future. After reading this chapter, you should have some idea of how to implement the test support of your choosing, and chances are good that someone out there has shared some matcher you might reuse.

The next section dips into the realm of concurrency. We already secretly used some concurrency features of Spring Integration to our advantage in tests, but now it's time to explore the different concurrency strategies. The combination of mocking and latching is especially powerful, so stay tuned!

18.3 Testing an asynchronous system

One of the trickiest things to solve cleanly in tests is assertions around related actions performed by multiple threads. A big advantage of staged event-driven architecture (SEDA; see chapter 2) is that components become passive and react to events rather than actively changing the world around them. This opens the door to decoupling cause and effect using a framework rather than having the complexities of asynchronous handoff emerge in business code. At runtime, though, these subtleties are essential to the proper functioning of the system. Therefore, they must be accounted for in tests. This section focuses on the concerns around testing an asynchronous system. An asynchronous system is a system in which multiple threads are involved in performing a bit of work (such as processing a message).

If a part of your system is designed to process messages asynchronously, you should keep an eye on certain things. As you saw in chapter 3, processing messages asynchronously can be done in several different ways. You can use a `<queue/>` element or a `<dispatcher/>` element. Also, when you use a publish-subscribe channel configured with a task executor, you're using asynchronous handoff. Finally, there are a few endpoints that can be configured with a `TaskExecutor` that will process a message in a different thread than the thread pushing the message in.

To give you a handle on this, remember that whenever an endpoint or channel is using storage or a task executor, it can cause asynchronous handoff.

Whenever asynchronous handoff is involved, *there are no chronological guarantees without explicit locking*. Luckily, getting explicit locking in place is simple in Spring Integration, but if you're not familiar with what happens under the hood, you can be sucked into hours of fruitless debugging.

18.3.1 Can't we wait for the message to come out the other end?

You sure can! In most cases, that's precisely what you should do. We look at a few exceptions later, but in the vast majority of cases, plugging into the output channel of your context and just waiting for the output message to arrive is sufficient. How do you go about it?

You might be expecting to see a loop with `Thread.sleep(..)` in there, or if you're more familiar with Java's concurrency support, you might expect a `CountDownLatch`. Things are even simpler than that.

For a standard test case, you just follow this simple recipe: make sure your output goes to a `QueueChannel` and receive from this channel before doing any assertions:

```
@Test
public void shouldInvokeService() {
    given(service.invoke(payload))
```

```
        .willReturn(newPayload);

    in.send(testMessage);
    Message m = out.receive(100);

    verify(service).invoke(payload);
    assertNotNull("Output didn't arrive!", m);
    assertThat(m, hasPayload(newPayload));
}
```

As you can see, you receive `m` from the output channel *before* you assert that the service has been invoked. Also, you use a timeout in the receive call to ensure the test doesn't run indefinitely. The assertions are done in the order that you expect them to succeed.

You're piggybacking on the contract of the receive method here. Because `receive` is a blocking call, you don't have to do any additional waiting to ensure a *happens-before* relationship between the service invocation and the verification. However complex your contexts get, it's almost always possible to find some output that will arrive only after the behavior you're trying to verify has been executed.

The timeout is also important. When designing a test case, you must understand that the main function of the test is to fail when the software doesn't behave correctly. This main function is best implemented when the failure clearly points out what part of the behavior was incorrect. If the `receive` call had no timeout, and an exception were thrown from the service, the test could run indefinitely. The test wouldn't fail in this case of malfunctioning in the software under test, so the test would be flawed. You could put a general timeout on the test to prevent it from running indefinitely. The timeout would fix the flaw, but the failure message would have no relation to the cause of the failure. The test would be correct but not very useful.

Finally, the assertions should be in the right order. If the service isn't invoked, you'll most likely get no output message. In this case, mixing the order of the assertions will make the test fail with an "Output didn't arrive" message, or worse, a `NullPointerException`. Thinking carefully about the order of the assertions can prevent this problem.

Before we look at exceptional cases in which waiting for output isn't an option or isn't sufficient to prove that the system functions correctly, we examine the need for proper test cases in asynchronous scenarios a bit further.

18.3.2 Avoiding the wicked ways of debugging

Before you get the wrong idea, let's make it clear that debugging is a skill that all excellent developers have and all novice developers should strive to learn. It's also the mother of all time wasters.

To tweak an old saying: Debug a program and you fix it for a day; improve the tests and the logging of a program, and you fix it for its lifetime. Before you dive into hours of debugging, you should ask yourself, what is this test not telling me that I need to know? The answer is often right in front of you, hard to reach with a debugger, easy to log. Once you have a test suite and some decent logging around the problem, another developer can continue where you left off. Better yet, in the

unthinkable scenario that the problem happens in production, you can ask the system administrator for the log file.

The reason we bring this up is because debugging is much harder in a concurrent scenario than in a single-threaded scenario. Spring Integration is inherently a concurrent framework, and if you have a concurrency-related bug lying dormant in your code, it might be awakened by wiring your service in a Spring Integration context. In single-threaded scenarios, debugging is great. It helps you understand the code more quickly than just reading through it would. In many cases, you don't want to fix all the logging in your program; you just want to see what's going on. That's fine, usually. But if multiple threads are entering the problem area of your code, debugging loses all its power. Suddenly, the debugger changes the timing that led to a race condition and often completely hides the bug from your sight. You could say that concurrency is debugger kryptonite.

Luckily, logging and test cases are much more reliable even when dealing with concurrent access. That's not to say that finding and analyzing a concurrency issue is easy. It's merely possible, and that's just about good enough. So heed this advice: especially when facing a concurrency bug, try to avoid the debugger and fix the problem with tests and logging. Now that you've learned to prefer testing and logging over the debugger in concurrent scenarios, you're ready to learn how to wait for messages to terminate inside an endpoint using latches and mocks.

18.3.3 Injecting latches into endpoints

Sometimes an endpoint has no output. As you read in chapter 4, these types of endpoints are called *channel adapters*. A channel adapter takes the payload of a message and feeds it to a service. This service may be a bean in your context, but it might also be a database, a web service, the filesystem, or standard output.

The tricky part is to wait for the invocation of this service before you start making assertions about the state of the system. In this section, we show you how you can inject latches into endpoints that have been mocked with Mockito. Similar strategies exist for other mocking frameworks, and the problem can also be solved with channel interceptors or AOP. Going over all these options is beyond the scope of this book, but this section should be enough to spark your imagination.

It's time to look back to our example. In figure 18.2, we showed how to mock out services from tests. That example required no latching inside the mock because you could just wait for the message to come out of the output channel, as discussed in the previous section. Let's explore an endpoint that's different in that respect.

When notifications are sent to the user, they're sent over an asynchronous communication channel. You specifically don't want to wait for the external system to confirm something was sent synchronously. That would block too many threads. Looking at the email outbound channel adapter, for instance, you need to confirm that a message reaches this adapter, but you don't need to test the sending of the email in this test. There should be another test for sending, but that's in the scope of infrastructure testing.

Let's say you want to test the following snippet:

```
<int:publish-subscribe-channel id="tripNotifications"
     datatype="siia.booking.domain.notifications.TripNotification"
     task-executor="taskScheduler"/>

<int:outbound-channel-adapter id="smsNotifier"
     channel="tripNotifications" ref="smsNotifierBean" method="notify"/>
```

You design your test to verify that a notification is passed into the `smsNotifierBean`'s `notify` method whenever a message containing the same notification is sent to the `tripNotifications` channel.

First you must make sure you replace the `smsNotifierBean` with a mock. You can use the same trick shown earlier:

```
<bean id="smsNotifierBean" class="org.mockito.Mockito"
      factory-method="mock">
    <constructor-arg
     value="siia.booking.domain.notifications.SmsNotifiable"/>
</bean>
```

Once that job is done, you can focus on the test itself.

The test sends a message containing the test notification to the channel. It then verifies that the method was invoked, but a simple verify call doesn't work here because the channel is an asynchronous publish-subscribe channel. You have to wait for the message to arrive before you can verify. This can be done using a latch injected into the mock:

```
private Answer countsDownLatch(final CountDownLatch notifierInvoked) {
    return new Answer() {
        @Override
        public Object answer(InvocationOnMock invocationOnMock)
                throws Throwable {
            notifierInvoked.countDown();
            return null;
        }
    };
}
```

With the answer returned by this method, you can tell Mockito to count down the latch passed in whenever a certain method is invoked. Let's go over the usage.

The JUnit test becomes

```
@Autowired
MessageChannel tripNotifications;

@Autowired
SmsNotifiable smsNotifier;

@Test
public void notificationShouldArriveAtSmsAdapter() throws Exception {
    TripNotification notification = mock(TripNotification.class);
    Message tripNotificationMessage =
            MessageBuilder.withPayload(notification)
                    .build();
```

```
    CountDownLatch notifierInvoked = new CountDownLatch(1);
    doAnswer(countsDownLatch(notifierInvoked))
            .when(smsNotifier).notify(notification);
    tripNotifications.send(tripNotificationMessage);
    notifierInvoked.await(100, MILLISECONDS);
    verify(smsNotifier).notify(notification);
}
```

Because the `notify` method returns void, you use Mockito's `doAnswer` method to record the behavior. You're essentially telling Mockito, "When the `notify` method is invoked on `smsNotifier`, react by counting down the `notifierInvoked` latch." Then it's a matter of awaiting the latch so you can execute assertions under the safe assumption that they'll happen after the message arrives at the endpoint.

Before we round up, we should give you some guidelines for making your applications easier to test. This isn't an easy thing, but it's a skill worth honing.

18.3.4 Structuring the configuration to facilitate testing

We can't overemphasize that changing the application to improve testability is a good thing. In Spring Integration applications, you usually see good decoupled code that's easy to test. But what about the configuration? With all that XML containing all those little SpEL expressions and intricate dependencies, you could easily get lost.

It's said that programming in XML is a bad thing (which it is). That's why Spring Integration focuses on XML as a domain-specific language for the *configuration* of enterprise integration patterns. It doesn't include logical constructs such as `<if>` or `<when>` in that domain-specific language. Nevertheless, it is arguably possible to cross the fuzzy line into XML programming if the configuration becomes too convoluted. This section offers some pointers to help you spot problems in this area and combat them with your test goggles on.

AVOID LOGIC IN XML

You can do complex things with Spring and Spring Integration, particularly using SpEL for elaborate routing. Don't! It might seem powerful, even simple at first, but testing logic that's embedded in XML is tough to the point of headache.

Instead, design your flows in linear steps as much as you can. If you want to use SpEL, keep it simple; delegate to Java code for the complex decisions. Also, it's fine to invoke methods on other objects from Java directly; not every fine-grained step needs to be a service activator.

SPLIT THE MAIN FLOW INTO SUBFLOWS

As your application gets larger, the configuration files grow too. At some point, it becomes hard to find that part of the configuration that you need to change. Take out the detailed flows and integrate them using import statements.

If you're used to Spring, you might've put configuration related to data access in a separate file, or you might've created several servlet contexts using the same root context. With a messaging application, splitting in layers isn't a good fit. It's better to divide the flow into different phases and give each phase its own context.

One way to make the subcomponents more testable is to define input and output channels in each subcontext and use bridges to glue them together in the main context. This way, a test that focuses on a particular subcontext in isolation can easily use the same concept to wire the input and output to channels that are specific to that test.

In the next section, we take a brief glimpse into the realm of threading.

18.3.5 How do I prove my code thread safe?

The short answer is that you don't. You can prove the correctness of your code under concurrent access, but it's usually unfeasible to test all possible concurrent scenarios and make sure they meet the specifications. But there are a few things you can do to help ensure your code is thread safe.

We don't go into great detail here, because concurrency is already discussed in detail in chapter 15. Just repeat to yourself: *Pass immutable objects between stateless services.*

Where testing is concerned, you can do your best to make sure concurrency bugs have a chance to surface in your test. For one, you should use *at least* the same number of threads in some of your integration tests as would be used in your production application. This ensures that the code is at least run concurrently in your continuous integration build. Some failures will still be unlikely to occur in a test, so this is by no means foolproof. Tests written this way might cause intermittent failures, which in many cases means you have a concurrency bug in your code.

Concurrency bugs are best tackled by logging and testing, but they can be a huge pain to reproduce. Some frameworks, such as *ConcuTest*, are helpful in provoking concurrency bugs by injecting yields and waits into your bytecode. If you learn these tools, you'll have a better chance of resolving concurrency bugs. In the future, Spring Integration's testing module may very well expand to include a full concurrency test suite.

18.4 Summary

In this chapter, we formalized our understanding of testing Spring Integration applications. First, we discussed the test support in Spring Integration's own test framework. Then, we detailed the strategies and rationale for mocking out external dependencies and business services from message flow tests. Finally, we discussed testing asynchronous applications on a broader level and showed you how to enforce chronological order in tests with asynchronous channels or mocks and latches. We also discussed thread safety.

Within the scope of the test framework, you saw different ways of matching messages, either by their headers or by their payloads. Matching payloads is helpful when you want to avoid unwrapping messages and casting their payloads to the expected types. We discussed support for unwrapping headers, including an example dealing with the whole map of headers.

We made a case for mocking business services out of tests. Because a Spring Integration configuration defines a message flow that's dynamically used at runtime, it

becomes important to test this configuration in relative isolation too. Mocking away external dependencies is an excellent way to achieve this goal.

Finally, we went into the details of testing a full asynchronous message flow. We explained how to use a blocking `receive` call to ensure chronological order in tests. Also we explained that when this doesn't work, you can use latches within mocks to enforce happens-before relationships.

This is the last chapter, but that doesn't mean it's least important. A proper understanding of how to test your application is both the end and the beginning of craftsmanship in software engineering.

JAVA/SPRING

Spring Integration IN ACTION

Fisher • Partner • Bogoevici • Fuld

Free eBook
SEE INSERT

Spring Integration extends the Spring Framework to support the patterns described in Gregor Hohpe and Bobby Woolf's *Enterprise Integration Patterns*. Like the Spring Framework itself, it focuses on developer productivity, making it easier to build, test, and maintain enterprise integration solutions.

Spring Integration in Action is an introduction and guide to enterprise integration and messaging using the Spring Integration framework. The book starts off by reviewing core messaging patterns, such as those used in transformation and routing. It then drills down into real-world enterprise integration scenarios using JMS, web services, filesystems, email, and more. You'll find an emphasis on testing, along with practical coverage of topics like concurrency, scheduling, system management, and monitoring.

What's Inside

- Realistic examples
- Expert advice from Spring Integration creators
- Detailed coverage of Spring Integration 2 features

This book is accessible to developers who know Java. Experience with Spring and EIP is helpful but not assumed.

Mark Fisher is the Spring Integration founder and project lead. **Jonas Partner**, **Marius Bogoevici**, and **Iwein Fuld** have all been project committers and are recognized experts on Spring and Spring Integration.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/SpringIntegrationinAction

"A wealth of good advice based on experience."
—From the Foreword by Rod Johnson
Founder of the Spring Framework

"Informative and well-written ... makes Spring Integration fun!"
—John Guthrie, SAP

"Bridges the gap between Spring and Enterprise Integration workspaces."
—Rick Wagner, Red Hat

"Comprehensive coverage of features and capabilities."
—Doug Warren, Java Web Services

"Spring Integration from its creators."
—Arnaud Cogoluègnes, coauthor of *Spring Batch in Action* and *Spring Dynamic Modules in Action*

ISBN 13: 978-1-935182-43-6
ISBN 10: 1-935182-43-9
54999
9 781935 182436

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]