Testing support

Spring Integration provides a number of utilities and annotations to help you test your application. Testing support is presented by two modules:

- `spring-integration-test-support` contains core items and shared utilities
- `spring-integration-test` provides mocking and application context configuration components for integration tests

`spring-integration-test-support` (`spring-integration-test` in versions before 5.0) provides basic, standalone utilities, rules, and matchers for unit testing. (it also has no dependencies on Spring Integration itself and is used internally in Framework tests). `spring-integration-test` aims to help with integration testing and provides a comprehensive high-level API to mock integration components and verify the behavior of individual components, including whole integration flows or only parts of them.

A thorough treatment of testing in the enterprise is beyond the scope of this reference manual. See the “Test-Driven Development in Enterprise Integration Projects” paper, by Gregor Hohpe and Wendy Istvanick, for a source of ideas and principles for testing your target integration solution.

The Spring Integration Test Framework and test utilities are fully based on existing JUnit, Hamcrest, and Mockito libraries. The application context interaction is based on the Spring test framework. See the documentation for those projects for further information.

Thanks to the canonical implementation of the EIP in Spring Integration Framework and its first-class citizens (such as `MessageChannel` , `Endpoint` and `MessageHandler`), abstractions, and loose coupling principles, you can implement integration solutions of any complexity. With the Spring Integration API for the flow definitions, you can improve, modify or even replace some part of the flow without impacting (mostly) other components in the integration solution. Testing such an integration solution is still a challenge, both from an end-to-end approach and from an in-isolation approach. Several existing tools can help to test or mock some integration protocols, and they work well with Spring Integration channel adapters. Examples of such tools include the following:

- Spring `MockMVC` and its `MockRestServiceServer` can be used for testing HTTP.
- Some RDBMS vendors provide embedded data bases for JDBC or JPA support.
- ActiveMQ can be embedded for testing JMS or STOMP protocols.

- There are tools for embedded MongoDB and Redis.
- Tomcat and Jetty have embedded libraries to test real HTTP, Web Services, or WebSockets.
- The `FtpServer` and `SshServer` from the Apache Mina project can be used for testing the FTP and SFTP protocols.
- Gemfire and Hazelcast can be run as real-data grid nodes in the tests.
- The Curator Framework provides a `TestingServer` for Zookeeper interaction.
- Apache Kafka provides admin tools to embed a Kafka Broker in the tests.

Most of these tools and libraries are used in Spring Integration tests. Also, from the GitHub repository (in the `test` directory of each module), you can discover ideas for how to build your own tests for integration solutions.

The rest of this chapter describes the testing tools and utilities provided by Spring Integration.

Testing Utilities

The `spring-integration-test-support` module provides utilities and helpers for unit testing.

TestUtils

The `TestUtils` class is mostly used for properties assertions in JUnit tests, as the following example shows:

```
@Test
public void loadBalancerRef() {
    MessageChannel channel = channels.get("lbRefChannel");
    LoadBalancingStrategy lbStrategy = TestUtils.getPropertyValue(channel,
                 "dispatcher.loadBalancingStrategy", LoadBalancingStrategy.class
    assertTrue(lbStrategy instanceof SampleLoadBalancingStrategy);
}
```

`TestUtils.getPropertyValue()` is based on Spring's `DirectFieldAccessor` and provides the ability to get a value from the target private property. As shown in the preceding example, it also supports nested properties access by using dotted notation.

The `createTestApplicationContext()` factory method produces a `TestApplicationContext` instance with the supplied Spring Integration environment.

See the Javadoc of other `TestUtils` methods for more information about this class.

Using the `SocketUtils` Class

The `SocketUtils` class provides several methods that select one or more random ports for exposing server-side components without conflicts, as the following example shows:

```
<bean id="socketUtils" class="org.springframework.util.SocketUtils" />

<int-syslog:inbound-channel-adapter id="syslog"
            channel="sysLogs"
            port="#{socketUtils.findAvailableUdpPort(1514)}" />

<int:channel id="sysLogs">
    <int:queue/>
</int:channel>
```

The following example shows how the preceding configuration is used from the unit test:

```
@Autowired @Qualifier("syslog.adapter")
private UdpSyslogReceivingChannelAdapter adapter;

@Autowired
private PollableChannel sysLogs;
...
@Test
public void testSimplestUdp() throws Exception {
    int port = TestUtils.getPropertyValue(adapter1, "udpAdapter.port", Integer.c
    byte[] buf = "<157>JUL 26 22:08:35 WEBERN TESTING[70729]: TEST SYSLOG MESSAG
    DatagramPacket packet = new DatagramPacket(buf, buf.length,
                              new InetSocketAddress("localhost", port));
    DatagramSocket socket = new DatagramSocket();
    socket.send(packet);
    socket.close();
    Message<?> message = foo.receive(10000);
    assertNotNull(message);
}
```

This technique is not foolproof. Some other process could be allocated the “free” port before your test opens it. It is generally more preferable to use server port `0`, let the operating system select the port for you, and then discover the selected port in your test. We have converted most framework tests to use this preferred technique.

Using `OnlyOnceTrigger`

`OnlyOnceTrigger` is useful for polling endpoints when you need to produce only one test message and verify the behavior without impacting other period messages. The following example shows how to configure `OnlyOnceTrigger`:

```
<bean id="testTrigger" class="org.springframework.integration.test.util.OnlyOnce

<int:poller id="jpaPoller" trigger="testTrigger">
    <int:transactional transaction-manager="transactionManager" />
</int:poller>
```

The following example shows how to use the preceding configuration of `OnlyOnceTrigger` for testing:

```
@Autowired
@Qualifier("jpaPoller")
PollerMetadata poller;

@Autowired
OnlyOnceTrigger testTrigger;
...
@Test
@DirtiesContext
public void testWithEntityClass() throws Exception {
    this.testTrigger.reset();
    ...
    JpaPollingChannelAdapter jpaPollingChannelAdapter = new JpaPollingChannelAda

    SourcePollingChannelAdapter adapter = JpaTestUtils.getSourcePollingChannelAd
                jpaPollingChannelAdapter, this.outputChannel, this.poller, this.
                this.getClass().getClassLoader());
    adapter.start();
    ...
}
```

Support Components

The `org.springframework.integration.test.support` package contains various abstract classes that you should implement in target tests

- `AbstractRequestResponseScenarioTests`

- `AbstractResponseValidator`
- `LogAdjustingTestSupport` (Deprecated)
- `MessageValidator`
- `PayloadValidator`
- `RequestResponseScenario`
- `SingleRequestResponseScenarioTests`

JUnit Rules and Conditions

The `LongRunningIntegrationTest` JUnit 4 test rule is present to indicate if test should be run if `RUN_LONG_INTEGRATION_TESTS` environment or system property is set to `true`. Otherwise it is skipped. For the same reason since version 5.1, a `@LongRunningTest` conditional annotation is provided for JUnit 5 tests.

Hamcrest and Mockito Matchers

The `org.springframework.integration.test.matcher` package contains several `Matcher` implementations to assert `Message` and its properties in unit tests. The following example shows how to use one such matcher (`PayloadMatcher`):

```
import static org.springframework.integration.test.matcher.PayloadMatcher.hasPay
...
@Test
public void transform_withFilePayload_convertedToByteArray() throws Exception {
    Message<?> result = this.transformer.transform(message);
    assertThat(result, is(notNullValue()));
    assertThat(result, hasPayload(is(instanceOf(byte[].class))));
    assertThat(result, hasPayload(SAMPLE_CONTENT.getBytes(DEFAULT_ENCODING)));
}
```

The `MockitoMessageMatchers` factory can be used for mocks for stubbing and verifications, as the following example shows:

```
static final Date SOME_PAYLOAD = new Date();

static final String SOME_HEADER_VALUE = "bar";

static final String SOME_HEADER_KEY = "test.foo";
...
Message<?> message = MessageBuilder.withPayload(SOME_PAYLOAD)
```

```
                    .setHeader(SOME_HEADER_KEY, SOME_HEADER_VALUE)
                    .build();
MessageHandler handler = mock(MessageHandler.class);
handler.handleMessage(message);
verify(handler).handleMessage(messageWithPayload(SOME_PAYLOAD));
verify(handler).handleMessage(messageWithPayload(is(instanceOf(Date.class))));
...
MessageChannel channel = mock(MessageChannel.class);
when(channel.send(messageWithHeaderEntry(SOME_HEADER_KEY, is(instanceOf(Short.cl
        .thenReturn(true);
assertThat(channel.send(message), is(false));
```

AssertJ conditions and predicates

Starting with version 5.2, the `MessagePredicate` is introduced to be used in the AssertJ `matches()` assertion. It requires a `Message` object as an expectation. And also ot can be configured with headers to exclude from expectation as well as from actual message to assert.

Spring Integration and the Test Context

Typically, tests for Spring applications use the Spring Test Framework. Since Spring Integration is based on the Spring Framework foundation, everything we can do with the Spring Test Framework also applies when testing integration flows.
The `org.springframework.integration.test.context` package provides some components for enhancing the test context for integration needs. First of all, we configure our test class with a `@SpringIntegrationTest` annotation to enable the Spring Integration Test Framework, as the following example shows:

```
@RunWith(SpringRunner.class)
@SpringIntegrationTest(noAutoStartup = {"inboundChannelAdapter", "*Source*"})
public class MyIntegrationTests {

    @Autowired
    private MockIntegrationContext mockIntegrationContext;

}
```

The `@SpringIntegrationTest` annotation populates a `MockIntegrationContext` bean, which you can autowire to the test class to access its methods. With the `noAutoStartup` option, the Spring Integration Test Framework prevents endpoints that are normally `autoStartup=true` from starting. The endpoints are matched to the provided patterns, which support the following simple pattern styles: `xxx*`, **`xxx`**, `*xxx`, and `xxx*yyy`.

This is useful when we would like to not have real connections to the target systems from inbound channel adapters (for example an AMQP Inbound Gateway, JDBC Polling Channel Adapter, WebSocket Message Producer in client mode, and so on).

The `MockIntegrationContext` is meant to be used in the target test cases for modifications to beans in the real application context. For example, endpoints that have `autoStartup` overridden to `false` can be replaced with mocks, as the following example shows:

```
@Test
public void testMockMessageSource() {
    MessageSource<String> messageSource = () -> new GenericMessage<>("foo");

    this.mockIntegrationContext.substituteMessageSourceFor("mySourceEndpoint", m

    Message<?> receive = this.results.receive(10_000);
    assertNotNull(receive);
}
```

The `mySourceEndpoint` refers here to the bean name of the `SourcePollingChannelAdapter` for which we replace the real `MessageSource` with our mock. Similarly the `MockIntegrationContext.substituteMessageHandlerFor()` expects a bean name for the `IntegrationConsumer`, which wraps a `MessageHandler` as an endpoint.

After test is performed you can restore the state of endpoint beans to the real configuration using `MockIntegrationContext.resetBeans()`:

```
@After
public void tearDown() {
    this.mockIntegrationContext.resetBeans();
}
```

See the Javadoc for more information.

Integration Mocks

The `org.springframework.integration.test.mock` package offers tools and utilities for mocking, stubbing, and verification of activity on Spring Integration components. The mocking functionality is fully based on and compatible with the well known Mockito Framework. (The current Mockito transitive dependency is on version 2.5.x or higher.)

MockIntegration

The `MockIntegration` factory provides an API to build mocks for Spring Integration beans that are parts of the integration flow (`MessageSource` , `MessageProducer` , `MessageHandler` , and `MessageChannel`). You can use the target mocks during the configuration phase as well as in the target test method to replace the real endpoints before performing verifications and assertions, as the following example shows:

```
<int:inbound-channel-adapter id="inboundChannelAdapter" channel="results">
    <bean class="org.springframework.integration.test.mock.MockIntegration" fact
        <constructor-arg value="a"/>
        <constructor-arg>
            <array>
                <value>b</value>
                <value>c</value>
            </array>
        </constructor-arg>
    </bean>
</int:inbound-channel-adapter>
```

The following example shows how to use Java Configuration to achieve the same configuration as the preceding example:

```
@InboundChannelAdapter(channel = "results")
@Bean
public MessageSource<Integer> testingMessageSource() {
    return MockIntegration.mockMessageSource(1, 2, 3);
}
...
StandardIntegrationFlow flow = IntegrationFlows
        .from(MockIntegration.mockMessageSource("foo", "bar", "baz"))
        .<String, String>transform(String::toUpperCase)
        .channel(out)
        .get();
IntegrationFlowRegistration registration = this.integrationFlowContext.registrat
        .register();
```

For this purpose, the aforementioned `MockIntegrationContext` should be used from the test, as the following example shows:

```
this.mockIntegrationContext.substituteMessageSourceFor("mySourceEndpoint",
        MockIntegration.mockMessageSource("foo", "bar", "baz"));
Message<?> receive = this.results.receive(10_000);
assertNotNull(receive);
assertEquals("FOO", receive.getPayload());
```

Unlike the Mockito `MessageSource` mock object, the `MockMessageHandler` is a regular `AbstractMessageProducingHandler` extension with a chain API to stub handling for incoming messages. The `MockMessageHandler` provides `handleNext(Consumer<Message<?>>)` to specify a one-way stub for the next request message. It is used to mock message handlers that do not produce replies. `handleNextAndReply(Function<Message<?>, ?>)` is provided for performing the same stub logic for the next request message and producing a reply for it. They can be chained to simulate any arbitrary request-reply scenarios for all expected request messages variants. These consumers and functions are applied to the incoming messages, one at a time from the stack, until the last, which is then used for all remaining messages. The behavior is similar to the Mockito `Answer` or `doReturn()` API.

In addition, you can supply a Mockito `ArgumentCaptor<Message<?>>` to the `MockMessageHandler` in a constructor argument. Each request message for the `MockMessageHandler` is captured by that `ArgumentCaptor`. During the test, you can use its `getValue()` and `getAllValues()` methods to verify and assert those request messages.

The `MockIntegrationContext` provides a `substituteMessageHandlerFor()` API that lets you replace the actual configured `MessageHandler` with a `MockMessageHandler` in the endpoint under test.

The following example shows a typical usage scenario:

```
ArgumentCaptor<Message<?>> messageArgumentCaptor = ArgumentCaptor.forClass(Messa

MessageHandler mockMessageHandler =
        mockMessageHandler(messageArgumentCaptor)
                .handleNextAndReply(m -> m.getPayload().toString().toUpperCase()

this.mockIntegrationContext.substituteMessageHandlerFor("myService.serviceActiva
                                mockMessageHandler);
GenericMessage<String> message = new GenericMessage<>("foo");
this.myChannel.send(message);
Message<?> received = this.results.receive(10000);
assertNotNull(received);
```

```
    assertEquals("FOO", received.getPayload());
    assertSame(message, messageArgumentCaptor.getValue());
```

See the `MockIntegration` and `MockMessageHandler` Javadoc for more information.

Other Resources

As well as exploring the test cases in the framework itself, the Spring Integration Samples repository has some sample applications specifically made to show testing, such as `testing-examples` and `advanced-testing-examples`. In some cases, the samples themselves have comprehensive end-to-end tests, such as the `file-split-ftp` sample.

Version 5.4.5
Last updated 2021-03-17 22:54:16 UTC