Suporte de teste

O Spring Integration fornece vários utilitários e anotações para ajudá-lo a testar seu aplicativo. O suporte de teste é apresentado por dois módulos:

- `spring-integration-test-support` contém itens principais e utilitários compartilhados
- `spring-integration-test` fornece simulação e componentes de configuração de contexto de aplicativo para testes de integração

`spring-integration-test-support` (`spring-integration-test` em versões anteriores a 5.0) fornece utilitários, regras e comparadores básicos e autônomos para teste de unidade. (ele também não tem dependências no próprio Spring Integration e é usado internamente em testes do Framework). `spring-integration-test` visa ajudar no teste de integração e fornece uma API abrangente de alto nível para simular componentes de integração e verificar o comportamento de componentes individuais, incluindo fluxos de integração inteiros ou apenas partes deles.

Um tratamento completo dos testes na empresa está além do escopo deste manual de referência. Consulte o artigo “Desenvolvimento orientado a testes em projetos de integração empresarial” , de Gregor Hohpe e Wendy Istvanick, para obter uma fonte de ideias e princípios para testar sua solução de integração de destino.

O Spring Integration Test Framework e os utilitários de teste são totalmente baseados nas bibliotecas JUnit, Hamcrest e Mockito existentes. A interação do contexto do aplicativo é baseada na estrutura de teste Spring . Consulte a documentação desses projetos para obter mais informações.

Graças à implementação canônica do EIP no Spring Integration Framework e seus cidadãos de primeira classe (como `MessageChannel` , `Endpoint` e `MessageHandler`), abstrações e princípios de acoplamento flexível, você pode implementar soluções de integração de qualquer complexidade. Com a API Spring Integration para as definições de fluxo, você pode melhorar, modificar ou até mesmo substituir alguma parte do fluxo sem impactar (principalmente) outros componentes na solução de integração. Testar essa solução de integração ainda é um desafio, tanto de uma abordagem de ponta a ponta quanto de uma abordagem isolada. Várias ferramentas existentes podem ajudar a testar ou simular alguns protocolos de integração e funcionam bem com adaptadores de canal Spring Integration. Exemplos de tais ferramentas incluem o seguinte:

- Spring `MockMVC` e its `MockRestServiceServer` podem ser usados para testar HTTP.

- Alguns fornecedores de RDBMS fornecem bancos de dados integrados para suporte JDBC ou JPA.
- ActiveMQ pode ser integrado para testar os protocolos JMS ou STOMP.
- Existem ferramentas para MongoDB e Redis incorporados.
- Tomcat e Jetty têm bibliotecas incorporadas para testar HTTP, Web Services ou WebSockets reais.
- O `FtpServer` e `SshServer` do projeto Apache Mina podem ser usados para testar os protocolos FTP e SFTP.
- Gemfire e Hazelcast podem ser executados como nós de grade de dados reais nos testes.
- O Curator Framework fornece um `TestingServer` recurso para interação com o Zookeeper.
- O Apache Kafka fornece ferramentas administrativas para incorporar um Kafka Broker nos testes.

A maioria dessas ferramentas e bibliotecas são usadas em testes de integração Spring. Além disso, no repositório GitHub (no `test` diretório de cada módulo), você pode descobrir ideias sobre como construir seus próprios testes para soluções de integração.

O restante deste capítulo descreve as ferramentas de teste e utilitários fornecidos pelo Spring Integration.

Utilitários de teste

O `spring-integration-test-support` módulo fornece utilitários e auxiliares para testes de unidade.

TestUtils

A `TestUtils` classe é usada principalmente para declarações de propriedades em testes JUnit, como mostra o exemplo a seguir:

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

`TestUtils.getPropertyValue()` é baseado no Spring `DirectFieldAccessor` e fornece a capacidade de obter um valor da propriedade privada de destino. Conforme mostrado no exemplo anterior, ele também oferece suporte ao acesso a propriedades aninhadas usando a notação pontilhada.

O `createTestApplicationContext()` método de fábrica produz uma `TestApplicationContext` instância com o ambiente Spring Integration fornecido.

Consulte o Javadoc de outros `TestUtils` métodos para obter mais informações sobre esta classe.

Usando a `SocketUtils` aula

A `SocketUtils` classe fornece vários métodos que selecionam uma ou mais portas aleatórias para expor componentes do lado do servidor sem conflitos, como mostra o exemplo a seguir:

```
<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>
```

O exemplo a seguir mostra como a configuração anterior é usada no teste de unidade:

```
@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);
}
```

Essa técnica não é infalível. Algum outro processo pode ser alocado na porta “livre” antes de seu teste abri-la. Geralmente é mais preferível usar a porta do servidor `0`, deixar o sistema operacional selecionar a porta para você e, em seguida, descobrir a porta selecionada em seu teste. Convertemos a maioria dos testes de estrutura para usar essa técnica preferida.

Usando `OnlyOnceTrigger`

`OnlyOnceTrigger` é útil para pontos de extremidade de polling quando você precisa produzir apenas uma mensagem de teste e verificar o comportamento sem impactar outras mensagens de período. O exemplo a seguir mostra como configurar `OnlyOnceTrigger`:

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

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

O exemplo a seguir mostra como usar a configuração anterior de `OnlyOnceTrigger` para teste:

```
@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();
    ...
}
```

Componentes de Suporte

O `org.springframework.integration.test.support` pacote contém várias classes abstratas que você deve implementar em testes de destino

- `AbstractRequestResponseScenarioTests`
- `AbstractResponseValidator`
- `LogAdjustingTestSupport` (Descontinuada)
- `MessageValidator`
- `PayloadValidator`
- `RequestResponseScenario`
- `SingleRequestResponseScenarioTests`

Regras e condições JUnit

A `LongRunningIntegrationTest` regra de teste JUnit 4 está presente para indicar se o teste deve ser executado se o `RUN_LONG_INTEGRATION_TESTS` ambiente ou propriedade do sistema estiver definido como `true`. Caso contrário, ele é ignorado. Pelo mesmo motivo desde a versão 5.1, uma `@LongRunningTest` anotação condicional é fornecida para testes JUnit 5.

Matchers Hamcrest e Mockito

O `org.springframework.integration.test.matcher` pacote contém várias `Matcher` implementações para afirmar `Message` e suas propriedades em testes de unidade. O exemplo a seguir mostra como usar um desses 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)));
}
```

A `MockitoMessageMatchers` fábrica pode ser usada para simulações para stubbing e verificações, como mostra o exemplo a seguir:

```
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));
```

Condições e predicados AssertJ

A partir da versão 5.2, o `MessagePredicate` é introduzido para ser usado na `matches()` asserção AssertJ . Requer um `Message` objeto como expectativa. E também não pode ser configurado com cabeçalhos para excluir da expectativa, bem como da mensagem real para afirmar.

Integração Spring e o contexto de teste

Normalmente, os testes de aplicativos Spring usam o Spring Test Framework. Como a integração do Spring é baseada no Spring Framework, tudo o que podemos fazer com o Spring Test Framework também se aplica ao testar os fluxos de integração. O `org.springframework.integration.test.context` pacote fornece alguns componentes para aprimorar o contexto de teste para as necessidades de integração. Em primeiro lugar, configuramos nossa classe de teste com uma `@SpringIntegrationTest` anotação para habilitar o Spring Integration Test Framework, como mostra o exemplo a seguir:

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

```
    @Autowired
    private MockIntegrationContext mockIntegrationContext;

}
```

A `@SpringIntegrationTest` anotação preenche um `MockIntegrationContext` bean, que você pode autowire para a classe de teste para acessar seus métodos. Com a `noAutoStartup` opção, o Spring Integration Test Framework evita que os terminais que normalmente são `autoStartup=true` iniciados. Os terminais são compatíveis com os padrões fornecidos, que suportam os seguintes estilos padrão simples: `xxx*` , **`xxx`** , `*xxx` , e `xxx*yyy` .

Isso é útil quando desejamos não ter conexões reais com os sistemas de destino de adaptadores de canal de entrada (por exemplo, um gateway de entrada AMQP, adaptador de canal de pesquisa JDBC, WebSocket Message Producer no modo cliente e assim por diante).

O `MockIntegrationContext` destina-se a ser usado nos casos de teste de destino para modificações nos beans no contexto real do aplicativo. Por exemplo, endpoints que foram `autoStartup` substituídos por `false` podem ser substituídos por simulações, como mostra o exemplo a seguir:

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

    this.mockIntegrationContext.substituteMessageSourceFor("mySourceEndpoint", m

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

O `mySourceEndpoint` refere-se aqui ao nome do bean de `SourcePollingChannelAdapter` para o qual substituímos o real `MessageSource` por nosso mock. Da mesma forma, o `MockIntegrationContext.substituteMessageHandlerFor()` espera um nome de bean para o `IntegrationConsumer` , que envolve a `MessageHandler` como um ponto de extremidade.

Após a realização do teste, você pode restaurar o estado dos beans do endpoint para a configuração real usando `MockIntegrationContext.resetBeans()`:

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

Consulte o Javadoc para obter mais informações.

Simulações de integração

O `org.springframework.integration.test.mock` pacote oferece ferramentas e utilitários para simulação, criação de stub e verificação de atividade nos componentes do Spring Integration. A funcionalidade de simulação é totalmente baseada e compatível com o conhecido Mockito Framework. (A dependência transitiva Mockito atual está na versão 2.5.x ou superior.)

Integração Mock

A `MockIntegration` fábrica fornece uma API para simulações de construção para o feijão de integração Primavera que são partes do fluxo de integração (`MessageSource`, `MessageProducer`, `MessageHandler` e `MessageChannel`). Você pode usar as simulações de destino durante a fase de configuração, bem como no método de teste de destino para substituir os endpoints reais antes de realizar verificações e asserções, como mostra o exemplo a seguir:

```
<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>
```

O exemplo a seguir mostra como usar a configuração Java para obter a mesma configuração do exemplo anterior:

```
@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();
```

Para tanto, `MockIntegrationContext` deve-se utilizar o supracitado a partir do teste, conforme mostra o exemplo a seguir:

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

Ao contrário do `MessageSource` objeto mock Mockito , o `MockMessageHandler` é uma `AbstractMessageProducingHandler` extensão regular com uma API em cadeia para manipulação de stub para mensagens recebidas. O `MockMessageHandler` fornece `handleNext(Consumer<Message<?>>)` a especificação de um stub unilateral para a próxima mensagem de solicitação. É usado para simular manipuladores de mensagens que não produzem respostas. `handleNextAndReply(Function<Message<?>, ?>)` é fornecido para executar a mesma lógica de stub para a próxima mensagem de solicitação e produzir uma resposta para ela. Eles podem ser encadeados para simular quaisquer cenários de solicitação-resposta arbitrários para todas as variantes de mensagens de solicitação esperadas. Esses consumidores e funções são aplicados às mensagens de entrada, uma por vez da pilha, até a última, que é então usada para todas as mensagens restantes. O comportamento é semelhante ao Mockito `Answer` ou `doReturn()` API.

Além disso, você pode fornecer um Mockito `ArgumentCaptor<Message<?>>` ao `MockMessageHandler` em um argumento do construtor. Cada mensagem de solicitação para o `MockMessageHandler` é capturada por isso `ArgumentCaptor` . Durante o teste, você pode usar seus métodos `getValue()` e `getAllValues()` para verificar e declarar essas mensagens de solicitação.

O `MockIntegrationContext` fornece uma `substituteMessageHandlerFor()` API que permite substituir o real configurado `MessageHandler` por um `MockMessageHandler` no endpoint em teste.

O exemplo a seguir mostra um cenário de uso típico:

```
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());
```

Consulte o `MockIntegration` e `MockMessageHandler` Javadoc para obter mais informações.

Outros recursos

Além de explorar os casos de teste na própria estrutura, o repositório Spring Integration Samples possui alguns aplicativos de amostra feitos especificamente para mostrar os testes, como `testing-examples` e `advanced-testing-examples`. Em alguns casos, as próprias amostras têm testes completos abrangentes, como a `file-split-ftp` amostra.

Versão 5.4.5
Última atualização em 2021-03-17 22:54:16 UTC