JAVA Mag

Java magazin™

Java | Architektur | Software-Innovation

JAVA 12

Alle neuen Features | Im Fokus: Switch Expressions
Überblick: Most Wanted JDKs | Infografik: JDK 12

Ausgabe 5.2019

Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

4 194586 709801 05

www.javamagazin.de

Spring Cloud Contract, Pact oder beides?

Consumer-driven Contracts mit Spring

In verteilten Systemen müssen Komponenten über externe Schnittstellen kommunizieren. Consumer-driven Contracts stellen einen speziellen Fall von Integrationstests dar. Sie ermöglichen es, bereits in der Entwicklung Schnittstellenverträge abzusichern, ohne dabei die beteiligten Services starten und End-to-end-Tests durchführen zu müssen. Für Spring-Entwickler stehen mit Pact JVM und Spring Cloud Contract gleich zwei Frameworks zur Verfügung, um solche Tests umzusetzen. Dieser Artikel soll bei der Entscheidung helfen, welches Framework man einsetzen möchte.

von Dr. Stephan Schuster

Das Ziel von Microservices ist, die Gesamtfunktionalität eines komplexen Systems in möglichst unabhängige Services aufzuteilen. Verglichen mit einem Monolithen möchte man dadurch Ziele wie ein schnelleres Deployment einzelner Services oder eine bessere und striktere Modularisierung des Systems erreichen. Da man es mit einem verteilten System zu tun hat, gehen diese Vorteile mit einer größeren technischen Komplexität einher. Einen Aspekt davon stellt die Kommunikation der Services untereinander über externe Schnittstellen wie REST APIs oder asynchrone Nachrichten dar.

Durch die externen Grenzen der Services untereinander sind Probleme beim Nutzen und Anbieten von Schnittstellen vorprogrammiert. So können z. B. neue Pflichtheader oder Parameter in einem REST Call dazukommen, welche die Schnittstelle brechen. Oft bemerken Konsumenten das erst spät, etwa bei End-to-end-Tests oder vielleicht sogar erst in der Produktion. Um die Entwicklung abzusichern, können Konsumenten das Verhalten einer Schnittstelle anhand von Mocks nachbilden und in Tests integrieren [1]. Dadurch wird sichergestellt, dass der Konsument (Consumer) den Dienst korrekt aufruft. Allerdings erfordert das, dass die Mocks aktuell gehalten werden. Die Tests müssen sich ebenfalls am Anbieter der Schnittstelle (Producer) und dessen Änderungen orientieren. Wird der Producer durch ein anderes Team bereitgestellt, kann dies das Aktuellhalten der Tests wegen geringerer direkter Kommunikation weiter erschweren.

Der Anspruch von Consumer-driven Contracts (CDC) ist, diese Lücke zwischen Consumer und Producer zu schließen, indem die Konsumenten dem Anbieter ihre Erwartungen in Form von Verträgen mitteilen. Dabei sind Verträge nicht im Sinne vollständiger Schnittstellenbeschreibungen (z. B. als WDSL) zu verstehen, sondern als minimale Anforderung eines Clients an einen API-Provider. Während der Anbieter eines API in der Regel Anforderungen mehrerer Clients befriedigen muss, sind diese nicht unbedingt daran interessiert, die komplette Schnittstelle zu kennen; oft reicht ein Ausschnitt eines JSON-Dokuments. Der Consumer-driven-Ansatz ist daher nur in Umfeldern interessant, in denen APIs parallel mit den Anforderungen der Konsumenten entwickelt werden.

Im Vorgehensmodell von CDC implementiert der Konsument zuerst gegen die zu definierende externe Schnittstelle und macht die daraus resultierende Anforderung dem Providerteam zugänglich. Der Konsument schreibt außerdem Tests, die die korrekte Verwendung des Vertrags durch den Consumer überprüfen. Clientseitig ist dieses Vorgehen identisch mit dem Testen gegen einen Mock-Server wie z. B. WireMock, der auf Aufrufe mit Stubs der externen Schnittstelle antwortet. Bei CDCs kommt nun aber hinzu, dass der Producer den Vertrag ebenfalls in Unit-Tests integriert. Diese vergleichen die Rückgabewerte der Schnittstelle mit der Erwartung des Vertrags. Erst wenn beide Tests erfolgreich sind, ist der Vertrag vollständig verifiziert. Weichen Werte auf der Providerseite ab oder hat sich die Verwendung der Schnittstelle auf Clientseite geändert, schlagen die Tests auf einer oder beiden Seiten fehl. Schnittstelle und ggf. Logik müssen angepasst werden. Somit hat man die Tests zweier unabhängiger Services miteinander integriert, ohne dass diese direkt miteinander kommunizieren müssen.

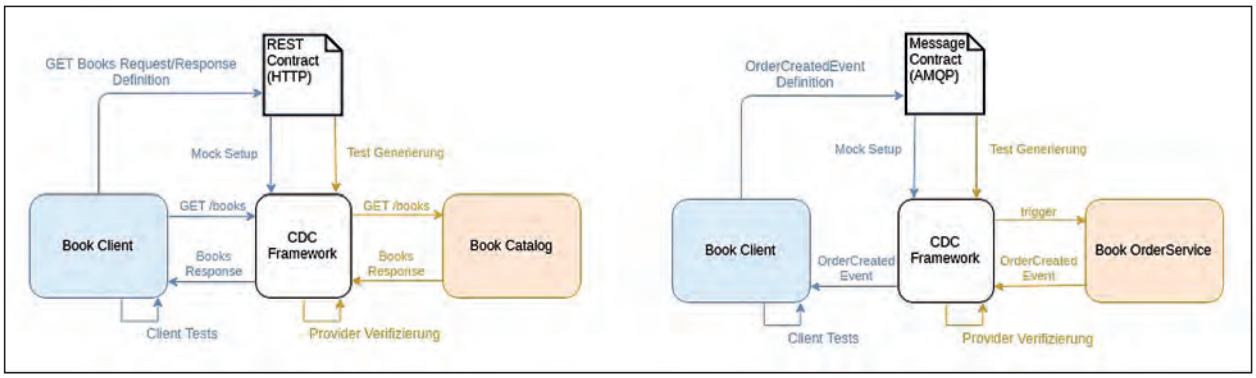

Abb. 1: Schematische Darstellung von CDC anhand der Beispielanwendung

Contract-Tests erhöhen also die Sicherheit zur Entwicklungszeit, da Fehler bei der formalen Verwendung der Schnittstellen schnell sichtbar werden und dann behoben werden können. Eine komplette Integration mit anderen Services ist an dieser Stelle nicht nötig. Contract-Tests können allerdings keine End-to-end-Tests ersetzen. Sie können als Integrationstests im Sinne der fowlerschen Testpyramide gesehen werden, deren Grenze zum (gemockten) externen System hin verläuft. Sie berücksichtigen daher auch nicht, ob die Daten der Stubs den realen Daten eines Produktivsystems entsprechen. Hierfür sind natürlich nach wie vor z. B. End-to-end-REST-API-Tests, UI-Tests oder Acceptance-Tests erforderlich [2].

Pact JVM [3] und Spring Cloud Contract (SCC) [4] sind zwei Frameworks, die CDC in der Java-Welt unterstützen. Dieser Artikel stellt diese Frameworks kurz vor und zeigt, wie man mit ihnen Tests in einer Spring-Anwendung schreibt. Der Artikel möchte eine etwas umfassendere Darstellung bieten und behandelt neben REST-API-Contract-Tests (über die bereits eine Reihe von Artikeln und Blogeinträgen existiert) auch Messaging-Schnittstellen. Dabei beschränken sich – bedingt durch das Kundenprojekt des Autors – Darstellung und Bewertung der Alternativen auf die Anwendung des jeweiligen Frameworks in einer Spring-Microservices-Landschaft. Als Beispiel dienen drei Microservices eines fiktiven Onlinebuchshops (das Beispiel ist auf GitHub verfügbar [5]:

- Ein Client-Service, der Anfragen von außen entgegennimmt und auf zwei interne Services weiterverteilt
- Ein Catalog Service für das Abrufen von Büchern mittels REST-Schnittstelle
- Ein Order Service für das Bestellen von Büchern über eine Messaging-Schnittstelle

Listing 1

```
@Service
public class BookRestUpstreamService  {
  private final RestTemplate restTemplate;
  public List<Book> getAllBooks() {
    String url = "http://localhost:8087/books";
    return Arrays.asList(restTemplate.exchange(url, HttpMethod.GET,
createHeaders(), Book[].class).getBody());
  }
  ...
}
@Service
public class BookOrderedEventRabbitListener {
  private final BookService bookService;
  @RabbitListener(queues = "order-queue")
  public void receiveMessage(BookOrderedEvent event) {
    bookService.orderConfirmed(event);
  }
}
```

Beispielanwendungsfall

Der Beispielclient steht exemplarisch für den Aufruf sowohl eines REST API als auch einer AMQP-Schnittstelle. Im REST-Fall wird eine einfache GET-Anfrage an den Catalog Service gestellt, um eine Liste von Büchern abzurufen; im Messaging-Fall wird ein *BookOrderedEvent* empfangen, nachdem ein Buch bestellt wurde. Diese über AMQP versendete Nachricht wird dabei von Spring bereits in ein *BookOrderedEvent* umgewandelt und dem *RabbitListener* übergeben. Listing 1 zeigt den REST-Client und den Message Listener der Beispielanwendung. **Abbildung 1** stellt schematisch den Ablauf von Consumer-driven-Contracts-Tests, abstrahiert vom jeweiligen Framework, für die beiden Fälle des Anwendungsbeispiels dar.

Pact

Ursprünglich in und für Ruby umgesetzt, existiert neben unterschiedlichen Implementierungen mit Pact JVM auch eine Variante, die auf der JVM nutzbar ist und einen JUnit Runner sowie Gradle- und Maven-Plug-ins zur Verfügung stellt. Darüber hinaus gibt es Spring-Erweiterungen, die eine Integration in Spring-Boot-Tests ermöglichen. Pact unterstützt CDC für REST-Schnittstellen ebenso wie für Messaging-Schnittstellen.

Ein Contract-Test wird in folgenden Schritten umgesetzt: Auf der Consumer-Seite wird in einem ersten Schritt der Vertrag per DSL definiert und in einem zweiten Testschritt verifiziert. In diesem Schritt wird ein

Mock-Server gestartet, der auf Anfragen mit den vorher definierten Antworten reagiert. Ein Server kann hierbei ein Mock-HTTP-Server oder ein Mock-AMQP-Server sein. Während der Ausführung des Tests wird eine Pact-Datei generiert und in Form eines JSON-Dokuments auf das Dateisystem geschrieben. Ein Verifikationsschritt ist also immer erforderlich, da nur so sichergestellt wird, dass der definierte Vertrag auch tatsächlich den Anforderungen des Clients gerecht wird (sonst könnte der Vertrag beliebig sein). Die Pact-Datei kann nun dem Producer übergeben werden, der dann die tatsächliche Antwort mit der erwarteten vergleichen kann. Die folgenden Abschnitte beschreiben diese Schritte im Detail.

Pact-Consumer-Test

Das Schreiben des Tests erfolgt unter Verwendung von pact-jvm-consumer-junit-Maven- oder Gradle-Plug-ins.

Listing 2

```
public class BookCatalogConsumerPactTest {
  @Rule
  public PactProviderRuleMk2 mockProvider =
    new PactProviderRuleMk2("book-catalog-service", "localhost", 8081, this);

  @Pact(consumer="books-client-catalog-rest-consumer")
  public RequestResponsePact pact(PactDslWithProvider builder) {
    DslPart payload = new PactDslJsonArray()
      .object()
      .integerType("id")
      .stringType("authorFirstName");
    return builder
      .given("get")
      .uponReceiving("A successful api GET call")
      .path("/books")
      .headers(expectedRequestHeaders())
      .method("GET")
      .willRespondWith()
      .status(200)
      .body(payload)
      .headers(responseHeaders())
      ...
      .toPact();
  }
  ...
  @Test
  @PactVerification
  public void verifyPact() {
    ResponseEntity<Book[]> response = new RestTemplate().
                exchange("http://localhost:8081/books", HttpMethod.GET,
                                                requestHeaders(), Book[].class);

    assertThat(response.getStatusCode().value(), equalTo(200));
    ...
  }
  ...
}
```

Listing 2 zeigt den vereinfachten Test für einen REST-API-Vertrag.

Pact JVM bietet neben der hier dargestellten annotations- und Rule-basierten Methode auch die Möglichkeit via Vererbung von *PactConsumerTest* mit ähnlichem Ergebnis, auf die hier allerdings nicht weiter eingegangen wird.

Bei der Durchführung des Tests wird ein Mock-HTTP-Server hochgefahren. Dieser beantwortet Anfragen mit den in Pact DSL definierten Objekten, die in der mit *@Pact* annotierten Methode erstellt werden. Die Anfragen werden dann mit der mit *@PactVerification* annotierten Methode gestellt und können dort mit JUnit-Mitteln auf Korrektheit geprüft werden. Listing 3 zeigt Ausschnitte der mittels (erfolgreichem) Test generierten Pact-Datei.

Die Pact-Datei definiert den erwarteten Response-Status und Response-Body. Die Felder des Bodys können über Matching Rules beschrieben werden, die z. B. den erwarteten Typ oder auch Muster mit Regular Expressions beschreiben. Ein REST-Vertrag kann dabei mehrere Interaktionen beinhalten, z. B. erwartete Antworten auf unterschiedliche Eingaben oder HTTP-Methoden. Die Zuordnung zur Testmethode erfolgt über den Namen des Providers (im Beispiel „book-catalog-service“, der optional auch in *@PactVerifier* angegeben werden kann). Optional können Zustände über das Element *providerStates* benannt werden, die den Zustand des Providers bei Aufruf der Methode angeben. Diese Zustände können später innerhalb des Providertests in eigenen Testmethoden hergestellt werden. Dadurch können auch Request-Response-Abfolgen simuliert

Listing 3

```
{
  "provider":{"name":"book-catalog-service"},
  "consumer":{"name":"books-client-catalog-rest-consumer"},
  "interactions":[
    {"description":"A successful api GET call",
      "request":{"method":"GET","path":"/books",
      },
      "response":{"status":200,
        "body":[
          {"id":1}
        ],
        "matchingRules":{
          "body":{
            "$[0].id":{
              "matchers":[{"match":"integer"}]
            }
          }
        }
      },
      "providerStates":[{"name":"get"}]
    }
  ]
}
```

werden, ohne dass eventuelle Zustände zwischen den Aufrufen gespeichert werden müssen.

Messaging Contracts werden analog erstellt. Listing 4 und Listing 5 zeigen Unit-Test und Pact-File für diesen Fall. Ein Messaging Contract kann ebenso wie ein API Contract mehrere Interaktionen enthalten. Komplexere Interaktionen können wieder über das State-Konzept abgebildet werden. Die Zuordnung zur Testmethode erfolgt hier allerdings nicht über den Producer-Namen, sondern über das *description*-Feld eines *Messages*-Eintrags. Auf diese Weise können analog mehrere zusammenhängende Nachrichten innerhalb eines Consumer-Tests definiert und verifiziert werden. Das unterscheidet sich vom REST-Fall, in dem alle Fälle innerhalb des Verifiers über mehrere HTTP Calls abgearbeitet werden können. Der Unterschied resultiert aus der *MessageMockProviderRule*, die das Simulieren des AMQP-Servers übernimmt. Diese Regel stellt keine unabhängige Umgebung zur Verfügung wie der Mock-HTTP-Server, sondern liefert lediglich die aus dem Pact generierte Nachricht als *byte[]* an die Testklasse zurück. Darüber hinaus besteht die – hier nicht dargestellte – Möglichkeit, über die *fragments*-Eigenschaft der *@PactVerification*-Annotation mehrere Nachrichten in eine Pact-Datei aufzunehmen.

Listing 4

```
public class BookOrderedEventConsumerPactTest {
  @Rule
  public MessagePactProviderRule mockProvider = new
                                        MessagePactProviderRule(this);
  private byte[] currentMessage;
  @Pact(provider = "book-order-service",consumer = "books-client-book-
                                                ordered-event-consumer")
  public MessagePact createPact(MessagePactBuilder builder) {
    PactDslJsonBody body = new PactDslJsonBody();
    body.stringType("customerId");
    return builder
      .given("orderCommandReceived")
      .metaData("destination": "order-exchange")
      .expectsToReceive("A message sent via order-exchange")
      .withContent(body)
      .toPact();
  }
  @Test
  @PactVerification({"book-order-service"})
  public void verify() throws Exception {
    byte[] message = mockProvider.getMessage();
    Assert.assertNotNull(new String(currentMessage));
  }
}
```

Listing 5

```
{"consumer":{"name":"books-client-book-ordered-event-consumer"},
  "provider":{"name":"book-order-service"},
  "messages":[{
    "description":"A message sent via order-exchange",
    "contents":{"isbn":"978-3-86680-192-9","customerId":"string"},
    "providerStates":[{"name":"orderCommandReceived"}],
    "matchingRules":{
      "body":{
        "$.customerId":{"matchers":[{"match":"type"}],
          "combine":"AND"
        },
        "$.isbn":{
          "matchers":[{"match":"regex","regex":"[0-9]{3}..."}],
          "combine":"AND"
        }
      }
    }
  }
]}
```

Pact-Producer-Test

Auf der Providerseite kann der generierte Vertrag nun getestet werden. Das setzt voraus, dass ein Server läuft, der die reale Antwort erzeugt, die mit dem Vertrag verglichen werden soll. Für den REST-Fall stehen mehrere Möglichkeiten zur Verfügung. Benutzt man den *PactRunner* JUnit Runner, erfolgt dies über ein Build-Tool wie Gradle oder Maven, das dafür sorgen muss, dass der Service-Provider gestartet wird. Alternativ kann man mit dem Plug-in pact-jvm-provider-spring den Server über einen Spring-Boot-Test ohne weitere Build-Konfiguration starten. Diese Möglichkeit ist in Listing 6 dargestellt. Im Unit-Test selbst wird mit dem *HttpTarget* ein Client genutzt, der einen laufenden Server an einer angegebenen Adresse erwartet. Die *@PactBroker*-Annotation gibt an, von wo der Vertrag des Providers *book-catalog-service* heruntergeladen werden kann. Das ist hier ein *PactBroker* (s. u.), könnte aber z. B. auch einfach ein lokales Verzeichnis sein. Sobald das geschehen ist, generiert der *PactRunner* aus dem Vertrag Unit-Tests, ruft über das *HttpTarget* die Schnittstelle auf und verifiziert das Ergebnis. Als Entwickler

Listing 6

```
@RunWith(SpringRestPactRunner.class)
@SpringBootTest
@Provider("book-catalog-service")
@PactBroker(host="localhost", port="80", protocol = "http")
public class BookCatalogRestEndpointPactOnlyTest {
  @TestTarget
  public final Target target = new HttpTarget(8087);

  @Test
  @State("get")
  public void verifySuccessfulGet() {
    ...
  }
}
```

muss man sich nur um die Konfiguration kümmern. Durch die *@State*-Annotation kann z. B. ein Zustand für diesen Testaufruf hergestellt werden, ähnlich wie in einem *@Before* eines üblichen JUnit-Tests. Pact ruft diese Methode vor der Verifikation auf. Sollen noch weitere Testschritte ausgeführt werden, kann das in derselben Methode erfolgen.

Bei einem Message-Contract ändert sich lediglich das Target zu einem AMQP Target. Als Nachteil erweist sich hier die fehlende Integration mit Spring. Zunächst müssen die beteiligten Services gemockt oder selbst instanziiert werden, da eine Ausführung als Spring-Boot-Test nicht möglich ist. Das AMQP Target ruft jede mit *@PactVerifyProvider* annotierte Methode auf und erwartet als Rückgabewert die JSON Payload, die von der Schnittstelle zurückgegeben wird. Auf diesen Wert wird dann analog zum REST-Fall die Verifizierung angewendet. Das bedeutet letztlich, dass der Producer nicht als Integrationstest ausgeführt werden kann, der z. B. auch die Zustellung der Nachricht an die korrekte Destination sicherstellt, sondern lediglich die Payload überprüfen kann. Ebenso eröffnet das auch die Möglichkeit, im Test selbst eine Antwort zu generieren, die nicht mit der tatsächlichen Ausgabe korrespondiert. Listing 7 zeigt einen Ausschnitt aus dem Unit-Test. Die Verbindung zur Pact-Datei wird über die *@PactVerifyProvider*-Annotation hergestellt, die dem *description*-Feld im Vertrag entsprechen muss. Optional kann hier wieder eine mit *@State* annotierte Methode hinzugefügt werden, die vor dem Test aufgerufen wird.

Pact Broker

Um auf der Producer-Seite einen Pact verifizieren zu können, müssen die Dateien zur Verfügung gestellt werden. Im einfachsten Fall kopiert man einfach die Dateien in das Build-Verzeichnis, z. B. indem man die Dateien in das Repository des Providers mit eincheckt.

Mit dem Pact Broker stellt das Pact-Projekt einen eigenen Server für den Austausch von Verträgen bereit. Consumer veröffentlichen ihre Verträge auf dem Server, Producer können sich jeweils die neueste Version herunterladen. Darüber hinaus bietet der Pact Broker nützliche Funktionen für die Integration in das Deployment: Pact-Versionen, die Versionen der Consumer und Producer sowie die Ergebnisse der Verifizierung werden im Broker gespeichert. So können Vertragsverletzungen direkt sichtbar gemacht und festgestellt werden, welche Client- und

Listing 7

```
@RunWith(PactRunner.class)
@Provider("book-order-service")
@PactBroker(host="localhost", port="80", protocol = "http")
public class OrderEventPactOnlyTest {
  @TestTarget
  public final Target target = new AmqpTarget(Collections.singletonList("de.sidion.books.order.*"));
  ...
  @Test
  @PactVerifyProvider("A message sent via order-exchange")
  public String verifyMessageForOrder() throws Exception {
    ...
    when(repo.createOrder(any(), any(), any()))
      .thenReturn(BookOrder.of(bookId, isbn, customerId));
       ArgumentCaptor<BookOrderedEvent> capt = ArgumentCaptor.forClass(BookOrderedEvent.class);
    service.createBookOrder("1", "1", "978-3-86680-192-9");
    Mockito.verify(notificationService).publishBookOrderedEvent(capt.capture());
    BookOrderedEvent event = capt.getValue();
    return new ObjectMapper().writeValueAsString(event);
  }
}
```

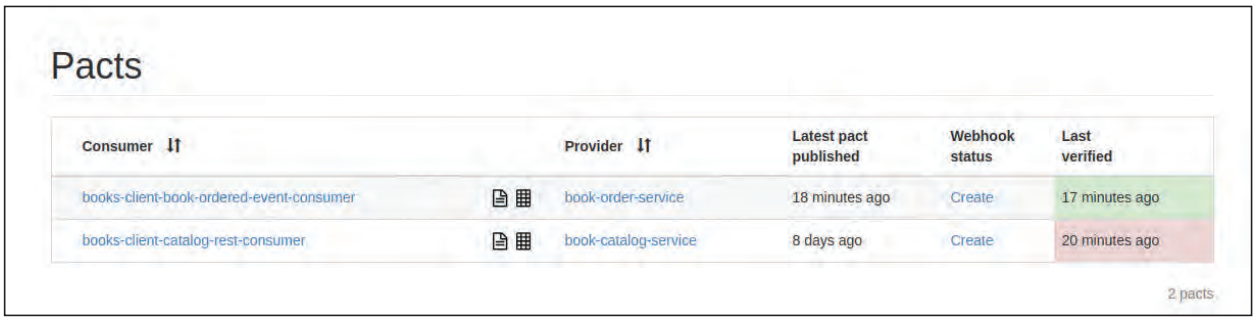

Abb. 2: Pact Broker mit den Verträgen der Beispielanwendung

Serverversionen kompatibel sind. Mittels API-Aufrufen können diese Features auch von automatisierten Delivery Pipelines genutzt werden (**Abb. 2**).

Spring Cloud Contract (SCC)

SCC entstand im Umfeld des Spring-Cloud-Projekts. Naturgemäß auf das Spring-Ökosystem ausgerichtet, bietet es aber die Möglichkeit, Verträge im YAML-Format zu spezifizieren und ein Spring Cloud Contract Docker Image zu nutzen. Das ermöglicht es SCC, auch Nicht-JVM-Sprachen zu nutzen [6]. Das ist keine wirklich polyglotte Lösung, die Implementierung basiert nach wie vor auf der JVM.

Ähnlich wie bei Pact erfolgt die Verifizierung auf Consumer-Seite in zwei Schritten: Zuerst wird der Vertrag definiert und dann im Rahmen eines Unit-Tests verifiziert. Anders als bei Pact JVM wird der Vertrag aber außerhalb des Unit-Tests in einer Groovy DSL oder YAML-Datei definiert. Der Consumer-Test enthält nur Code zur Validierung. Dabei startet der Stub Runner die Mock-Umgebung (WireMock für REST- und eine eigene AMQP-Mock-Implementierung für Messageverträge) und nutzt die Verträge, um entsprechende Antworten zu erzeugen. Auf der Producer-Seite vergleicht der Contract Verifier die Antwort mit den erwarteten Ergebnissen des Vertrags. Auch hier muss kein Test geschrieben, sondern lediglich die Testumgebung konfiguriert werden. Ermöglicht wird das durch Einbinden der entsprechenden Plug-ins, z. B. des spring-cloud-contract-gradle-plugin.

Der von SCC präferierte Workflow geht davon aus, dass Verträge vom Consumer erstellt, an das Producer-Team übermittelt (z. B. als Teil eines Pull Requests) und in dessen Repository gepflegt werden. Der Producer Build erstellt beim Publizieren der Artefakte ein zusätzliches JAR mit den Verträgen. Dafür müssen das Maven Publish Plugin und ein Maven Repository verwendet werden. Dieses Artefakt wird dann sowohl vom Client als auch vom Provider genutzt. Im REST-Fall enthält das JAR bereits WireMock Stubs, die aus den Verträgen generiert wurden. Im AMQP-Fall verwendet das eigens implementierte Mock-Framework die Vertragsdaten des Message Contracts.

Listing 8 und Listing 9 zeigen Verträge für REST und AMQP in der Groovy DSL. Ähnlich wie Pact REST Contracts definiert der Vertrag Response-Codes, Headerfelder und entsprechende Patterns für den Response-Body.

Die Payload eines Message Contracts wird nach denselben Prinzipien überprüft. Darüber hinaus unterscheidet SCC drei Modi: das Triggern der Nachricht durch eine Triggernachricht; zeitgesteuert, z. B. durch einen Cronjob; oder über eine Methode, wie im Beispiel dargestellt.

SCC-Consumer-Test

Listing 10 zeigt den Consumer-Test für eine REST-Schnittstelle. Über die *@AutoConfigureStubRunner-*

Listing 8

```
org.springframework.cloud.contract.spec.Contract.make {
  request {
    method 'GET'
    url '/books'
    headers {contentType('application/json')}
  }
  response {
    status 200
      body([["authorFirstName":$(regex("[a-zA-Z]*"))]])
      headers {
        contentType('application/json')
      }
  }
}
```

Listing 9

```
Contract.make {
  description("Example messaging contract")
  label 'book-order'
  input {
    triggeredBy('bookOrdered()')
  }
  outputMessage {
    sentTo 'order-exchange'
    body(["customerId": $(regex("[0-9]*"))])
    headers {
      messagingContentType(applicationJson())
    }
  }
}
```

Annotation wird angegeben, wo und welche Verträge für den Stub Runner benutzt werden sollen. Der restliche Test kann wie ein normaler Integrationstest geschrieben werden, z. B. als Spring-Boot-Test. Wird nun die externe REST-Schnittstelle aufgerufen, antwortet der durch den Stub Runner gestartete WireMock-Server. Da dieser in einem Zufallsport initialisiert wird, kann über *minPort* und *maxPort* die Range eingeschränkt werden. Im Beispiel soll es genau Port 9999 sein, der vom REST-Client der Anwendung verwendet wird.

Für Messaging Contracts stellt SCC eine eigene Mock-Implementierung zur Verfügung. In dieser Implementierung werden RabbitMQ-Mock-Komponenten wie Templates und Listener-Container definiert und mittels *ContractVerifierAmqpAutoConfiguration* dem Spring Context übergeben. Im Consumer-Test (Listing 11) wird dann durch Aufruf von *StubTrigger.trigger()* eine Nachricht an den Mock-AMQP-Server geschickt. Auf diese Weise kann ein Integrationstest mit komplettem Spring Context gestartet werden, ohne von externer Kommunikationsinfrastruktur abhängig zu sein.

Listing 10

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureStubRunner(stubsMode = StubRunnerProperties.StubsMode.LOCAL,
minPort = 9999, maxPort = 9999,
ids = "de.sidion.books:books-catalog-service-scc-only-test")
public class BookCatalogConsumerSccStubrunnerTest {
  @Autowired
  BookRestUpstreamService bookService; //Ruft das Catalog API auf (Listing 1)
  @Test
  public void verifyBookCatalogGetAllBooksContract() throws Exception {
    List<Book> books = bookService.getAllBooks();
    assertThat(books, Matchers.notNullValue());
    ...
  }
}
```

Listing 11

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureStubRunner(stubsMode = StubRunnerProperties.StubsMode.LOCAL, ids =
"de.sidion.books:books-order-service-scc-only-test")
public class BookOrderedEventConsumerSccStubrunnerTest {
  @Autowired StubTrigger stubTrigger;
  @Autowired BookService service; //wird vom BookOrderedEventRabbitListener
aufgerufen (Listing 1)
  @Test
  public void verifyBookOrderedEventContract() throws Exception {
    int counterBefore = service.getMessagesReceivedCounter();
    stubTrigger.trigger("book-order");
    int counterAfter = service.getMessagesReceivedCounter();
    assertThat(counterAfter, equalTo(counterBefore + 1));
  }
}
```

SCC-Producer-Test

Analog zur Vorgehensweise bei Pact besteht der Test auf Producer-Seite im Herunterladen des Vertrags und der Überprüfung der Erwartungen mit der tatsächlichen Ausgabe der Schnittstelle. Auch hier erfolgt die Überprüfung automatisch. Anders als bei Pact wird dabei jedoch Testcode generiert und im Build-Verzeichnis abgelegt. Als Entwickler muss man eine abstrakte Basistestklasse schreiben und diese über das Build-Skript dem Framework bekannt machen, sowie den Testklassenpfad erweitern. Die generierten Tests erweitern dann die Basisklasse. Listing 12zeigt den Ausschnitt aus dem Gradle-Build-File.

In Listing 12 wird auch offensichtlich, wie SCC die Maven-Nomenklatur verwendet, um Verträge im Repository zu finden. Diese Nomenklatur muss auch bei anderen Mechanismen – Git und Pact-Broker werden ebenfalls unterstützt – beibehalten bleiben.

Für die Verifizierung eines REST-API-Vertrags setzt SCC auf *MockMvc*-Klassen auf. Daher muss ein entsprechender Mock konfiguriert werden. Listing 13 zeigt das unter Verwendung von *RestAssured*.

Listing 14 zeigt den Test für einen Messaging Contract. Bei Messaging Contracts muss die Basistestklasse dafür sorgen, dass der im Vertrag angegebene Trigger ausgelöst wird. SCC erwartet, dass die Methode, die im Vertrag unter *triggeredBy* (Listing 9) angegeben wurde, in der Testklasse existiert, und ruft diese zu Beginn der Verifikation auf. Im Beispiel heißt diese Triggermethode, über welche der Producer die zu verifizierende Nachricht verschickt, *bookOrdered()*. Hier ist zu beachten, dass der SCC Verifier über *@AutoConfigureMessageVerifier* eingebunden werden muss. Der generierte

Listing 12

```
contracts {
  contractDependency {
    stringNotation = "${project.group}:book-catalog-service:+"
  }
  contractsPath = "*"
  baseClassForTests =
    "de.sidion.books.catalog.contracts.
BookCatalogRestEndpointBaseSccOnlyTest"
}
sourceSets {
  contractTest {
    java {
    compileClasspath+=main.output+test.output
    runtimeClasspath+=main.output+test.output
    srcDir file('build/generated-test-sources')
    }
  }
}
```

Test kann dann die an den Mock Exchange versendete Nachricht abfangen und verifizieren. Intern geschieht dies über die Verwendung von Mockito Spies in den Komponenten der *MessageVerifier*-Konfiguration.

Listing 15 verdeutlicht beispielhaft, wie der generierte Test über die Klasse *ContractVerifierMessaging* das *BookOrdered*-Event abfängt und verifiziert.

Vergleich von Pact und Spring Cloud Contract

Obwohl beide Ansätze Consumer-driven Contracts implementieren, werden doch einige Unterschiede offensichtlich:

- Pact ist in unterschiedlichen Implementierungen verfügbar. Dies ist ein Vorteil, hat aber auch eine Kehrseite: Die Integration mit Spring ist nicht komplett. Mit pact-jvm-provider-spring gibt es zwar ein Plug-in für Spring, allerdings noch nicht für Message Contracts.
- Bei Pact erschwert insbesondere die nur rudimentäre Unterstützung von Messaging-Verträgen das Testen, da hier eigentlich nur reine Stub-Tests mit Mockito oder ähnlichen Mitteln geschrieben werden können. Man schreibt hier keinen Integrationstest, der das Verhalten der Messaging-Komponenten wie Queues und Listener mittestet, sondern verifiziert lediglich die Payload. Orientiert man sich an der anfangs erwähnten Testpyramide, wird die Teststrategie inkonsistent.
- Bei SCC muss der Vertrag als Datei oder Skript außerhalb des JUnit-Tests gepflegt werden. Pact JVM hingegen bietet ein einfaches API, mit dem die Contract-Dateien programmatisch erzeugt werden können. In den Augen des Autors erleichtert dies das Schreiben und Pflegen des Vertrags, da Vertrag und Verifizierung Teil derselben Testklasse sind.
- Pact bietet mit dem Pact Broker ein spezielles Repository für Verträge sowie ein dazugehöriges API. Das vereinfacht die Pflege der Verträge, die automatisch über Build Tasks an den Broker übermittelt werden. Darüber hinaus vereinfacht es die Integration in Deployment-Pipelines: Beim Veröffentlichen eines neuen Vertrags kann der Provider automatisch validiert werden. Neben der Testabsicherung beim Build kann außerdem beim Deployment sichergestellt werden, dass nur kompatible Producer- und Consumer-Versionen installiert werden (das Fehlschlagen von Producer-Tests verhindert nicht, dass der Consumer Build erfolgreich ist und ggf. automatisch deployt wird).

Kombination beider Frameworks im Kundenprojekt

Als Praxisbeispiel kann ein Kundenprojekt dienen. Dabei handelt es sich um eine E-Commerce-Plattform, die mit Spring Boot Microservices implementiert ist. Ein Großteil der internen Kommunikation ist asynchron mit RabbitMQ umgesetzt, einige Schnittstellen mit REST. Für CDC sind derzeit noch keine externen Clients vorgesehen, eine Anbindung soll aber möglich sein. Die Verwendung von SCC bietet sich also gera-

Listing 13

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = BookCatalogApplication.class)
public abstract class BookCatalogRestEndpointBaseSccOnlyTest {
  @Autowired BookRestEndpoint endpoint;
  @Before
  public void setup() {
    RestAssuredMockMvc.standaloneSetup(endpoint);
  }
}
```

Listing 14

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMessageVerifier
public abstract class OderEventSccOnlyTest {
  @Autowired BookOrderDomainService service;
  protected void bookOrdered() {
    service.createBookOrder("1", "1", "978-3-86680-192-9");
  }
}

//BookOrderDomainService:
public void createBookOrder(String customerId, String bookId, String isbn) {
  BookOrder order = orderRepository.createOrder(customerId, bookId, isbn);
  BookOrderedEvent  event = BookOrderedEvent.of(order);
  rabbitTemplate.convertAndSend("order-exchange", "orders.books.#", event);
}
```

Listing 15

```
public class ContractVerifierTest extends OderEventSccOnlyTest {
  @Inject ContractVerifierMessaging contractVerifierMessaging;
  @Inject ContractVerifierObjectMapper contractVerifierObjectMapper;
  @Test
  public void validate_bookMessagingContract() throws Exception {
    // when:
    bookOrdered();
    // then:
    ContractVerifierMessage response = contractVerifierMessaging.receive("order-exchange");
    assertThat(response).isNotNull();
    assertThat(response.getHeader("contentType")).isNotNull();
      ...
    // and:
     DocumentContext parsedJson = JsonPath.parse
      (contractVerifierObjectMapper.writeValueAsString(response.getPayload()));
      assertThatJson(parsedJson).field("['customerId']").matches("[0-9]*");
    ...
  }
}
```

dezu an, insbesondere aufgrund der guten Unterstützung von AMQP. Andererseits bietet ein Pact Broker neben der Infrastrukturunterstützung weitere Vorteile, wie Offenheit für weitere Nicht-JVM-Consumer. Daher wurde entschieden, die hier als Vorteile beschriebenen Features beider Frameworks zu nutzen, indem auf der Consumer-Seite Pact und auf der Provider-Seite SCC verwendet wird.

Ermöglicht wird dies über das Plug-in spring-cloud-contract-pact von Spring Cloud Contract, mit dem Pact-Verträge in SCC-Verträge und umgekehrt übersetzt werden können. Während die Struktur von Pact und SCC-Verträgen sehr ähnlich ist und die Übersetzung problemlos funktioniert, ergeben sich starke Abweichungen für Messaging Contracts. So kennt Pact zum Beispiel keine Möglichkeit, die Message Destination zu definieren. SCC Message Contracts erwarten darüber hinaus eine Triggermethode, für die es ebenfalls kein Konzept in Pact gibt. Diese wird vom *providerStates*-Feld des Message Pacts gemappt. Die Message Destination wird (ab der neuesten Releaseversion 2.1.0, die seit Januar 2019 verfügbar ist) in einem speziellen *sentTo*-Metadateneintrag in der Pact-Datei erwartet.

Ein weiterer Nachteil dieser Art von Verwendung von SCC auf Providerseite ist, dass SCC das Ergebnis der Verifikation nicht von allein an den Pact Broker schickt. Für das Publizieren des Producer-Testergebnisses kann eine JUnit Rule implementiert werden, die das Ergebnis über das Pact Broker API an den Broker schickt (ebenfalls Teil des GitHub-Beispiel-Repositorys [5]).

Für das Kundenprojekt wurde folgende Vorgehensweise gewählt:

- Consumer Contracts werden mittels pact-jvm als JUnit-Test geschrieben und verifiziert.
- Alle generierten Verträge werden bei der Ausführung als Jenkins-Job mit dem Plug-in pact gradle auf den Pact Broker publiziert.
- Der WebHook des Pact Brokers ruft bei geänderten Verträgen den betroffenen Jenkins-Job-Provider auf. Dieser führt (unter anderem) die Verifizierung des Producers aus und publiziert das Ergebnis an den Broker.
- Wenn der Vertrag auf Providerseite nicht validiert werden kann, schlägt der entsprechende Jenkins-Job fehl. Das erzwingt die Anpassung der Implementierung auf Producer-Seite.
- Ein automatisiertes Deployment findet nur statt, wenn Consumer und Producer den aktuellsten Vertrag erfolgreich verifiziert haben. Hierfür muss neben den oben erwähnten Besonderheiten die Build-Tool-Konfiguration auf Producer-Seite angepasst werden (auf die Deployment Pipeline wird hier nicht eingegangen). Neben der Einbindung des Plug-ins muss im Contracts-Abschnitt nur der Pact Broker als Source für die Verträge konfiguriert werden, wie in Listing 16 dargestellt.

Listing 16

```
contracts {
  contractDependency {
    stringNotation = "${project.group}:book-order-service:+"
  }
  contractsPath = "*"
  contractsMode = "REMOTE"
  baseClassForTests = "de.sidion.books.order.contracts.OrderEventPactWithSccTest"
  contractRepository {
    repositoryUrl = "pact://http://localhost:80"
  }
}
```

Fazit

In diesem Artikel wurden die Funktionsweise und die Vorteile von SCC und Pact JVM demonstriert. Kurz zusammengefasst kann man den Vorteil von SCC in seiner Integration in das Spring-Ökosystem sehen, den Vorteil von Pact in seiner größeren Plattformunabhängigkeit und in Tools wie dem Pact Broker. Dennoch kann man mit etwas eigenem Aufwand beide Welten miteinander verbinden. Natürlich spielen die konkreten Anforderungen des Projekts eine Rolle. Möchte man z. B. in einem Spring-System keinen Pact Broker nutzen, ist es sinnvoller, nur SCC zu nutzen. Werden nur REST APIs verwendet, ist es praktisch kein Unterschied, welches Framework man wählt. Möchte man primär eine Lösung, um über RabbitMQ verschickte Nachrichten zu verifizieren, wäre SCC sicherlich auch hier das Mittel der Wahl. Kommen weitere Clients in Spiel, etwa JavaScript-Anwendungen, kann es sich lohnen, über eine Pact-Lösung oder ähnliche Kombinationen wie die hier vorgestellte nachzudenken.

Im Beispielprojekt auf GitHub [5] sind Implementierungen für jeden der hier dargestellten Fälle enthalten: Pact-Pact, SCC-SCC und Pact-SCC, jeweils für REST APIs und Message Contracts.

Dr. Stephan Schuster arbeitet als Architekt und Entwickler bei sidion und entwickelt seit über zehn Jahren Webanwendungen. Sein aktueller Schwerpunkt liegt auf Microservices-Architekturen.

Links & Literatur

[1] https://martinfowler.com/bliki/ContractTest.html

[2] https://martinfowler.com/articles/practical-test-pyramid.html

[3] https://docs.pact.io/

[4] https://spring.io/projects/spring-cloud-contract

[5] https://github.com/schustes/sidion-contract-examples

[6] https://spring.io/blog/2018/02/13/spring-cloud-contract-in-a-polyglot-world