Pacta sunt servanda

Consumer-Driven Contracts – Testen von Schnittstellen innerhalb einer Microservices-Architektur

Michael Vitz

In einer Microservices-Architektur entstehen viele Services – potenziell in den verschiedensten Programmiersprachen. Um eine reibungslose Kommunikation zwischen diesen zu gewährleisten, müssen die Schnittstellen passen und auch über längere Zeit stabil bleiben. Consumer-Driven Contracts stellt hierzu einen Ansatz dar, der zudem die Schnittstellen und deren Aufrufer noch zusätzlich testet.

Beschäftigt man sich mit aktuellen Themen rund um Softwarearchitektur, so kommt man um Microservices und den Hype um diese nicht herum.

Eine auf Microservices basierende Architektur ermöglicht eine unabhängige Deploybarkeit der einzelnen Services und gibt dem Entwickler die Freiheit, für jeden Service die passende Programmiersprache wählen zu können.

Diese Vorteile erkauft man sich jedoch damit, dass man von nun an ein verteiltes System entwickelt. Entwickler werden deshalb mit anderen Herausforderungen konfrontiert als in einem Monolithen.

Service-Kommunikation zwischen Microservices

Zerteilt man eine Anwendung in mehrere Services, müssen sich diese Services gegenseitig aufrufen. Idealerweise versucht man zwar, diese Aufrufe auf ein Minimum zu reduzieren, komplett vermeiden lassen sie sich aber zumeist nicht.

Der Grund, wieso man versucht, diese Aufrufe zu minimieren, ist einfach: Jeder Aufruf birgt die Gefahr, dass der aufgerufene Service aktuell nicht zur Verfügung steht, und vermindert somit auch die Verfügbarkeit unseres Services. Zudem sind solche Aufrufe zumindest im Vergleich zu lokalen Aufrufen langsam.

Damit die verbleibende Kommunikation reibungslos abläuft, muss zudem sichergestellt werden, dass beide beteiligten Services dieselbe Sprache sprechen. In Zeiten der Service-orientierten Architekturen (SAO) wurde dies über WSDL- und XML-Schemas sichergestellt. Heutzutage ist meistens REST mit JSON oder Messaging das Mittel der Wahl. Aber auch dabei muss dafür gesorgt werden, dass Änderungen einer Schnittstelle die Clients nicht brechen und andersherum die Clients die richtigen Abfragen durchführen. Häufig wird zu diesem Zweck auf Dokumentation, manuelle oder generierte, gesetzt.

Mit Consumer-Driven Contracts gibt es allerdings bereits seit geraumer Zeit ein Konzept, das mit Hilfe von ausführbaren Tests sicherstellt, dass Service-Anbieter und -Nutzer zusammenpassen.

Consumer-Driven Contracts?

Consumer-Driven Contracts [CDC] zeichnen sich, wie bereits am Namen erkennbar, dadurch aus, dass der Nutzer einer Service-Schnittstelle den Contract der Schnittstelle spezifiziert. Vorteilhaft hieran ist, dass die Aufrufer einer Schnittstelle besser wissen, wie sie diese nutzen. Hierdurch werden Annahmen, die ansonsten von den Entwicklern des Services getroffen werden, eliminiert.

Die so erstellten Contracts können anschließend automatisiert gegen die Schnittstelle geprüft werden. Somit erkennt man, ob der Service die Schnittstelle erfüllt. Zudem kann der Service die Schnittstelle auch ändern und weiterentwickeln, solange er die Contracts weiter erfüllt. Änderungen, die nicht rückwärts kompatibel sind, werden somit schnell gefunden. Somit kann man sich immer, wenn diese auftreten, in Ruhe Gedanken darüber machen, wie man mit diesen umgeht.

Pact

Mit Pact [Pact] gibt es eine Implementierung von Consumer-Driven Contracts, die auch auf der JVM nutzbar ist. Neben der Definition von HTTP-API-Contracts wird, seit Version 3, auch Messaging unterstützt.

Das Prinzip von Pact basiert auf den beiden Schritten „Definieren“ und „Verifizieren“ (s. Abb. 1) und nutzt hierzu das von Martin Fowler beschriebene Prinzip IntegrationContractTest [ICT].

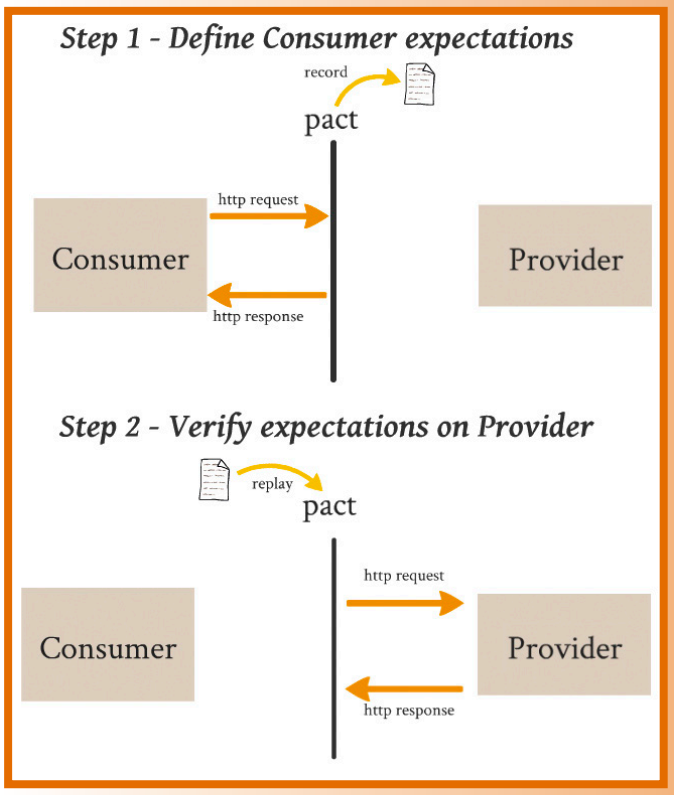

Abb. 1: Die zwei Schritte von Pact [PactParts]

Im ersten Schritt definiert der Aufrufer (Consumer) die von ihm durchzuführenden Aufrufe und die erwarteten Antworten auf diese. Dies geschieht innerhalb eines Unittests in der Programmiersprache, in der der Consumer geschrieben wird.

Führt man diese Tests aus, sorgt Pact dafür, dass ein Server gestartet wird, der anschließend auf die Aufrufe aus den Tests mit den vorher definierten Antworten antwortet. Soweit könnte man diese Tests auch mittels Mocks/Stubs schreiben und sich den nun wirklich stattfindenden Netzwerkverkehr sparen. Pact generiert jedoch zusätzlich während der Testausführung eine „pact"-Datei. Diese enthält alle zuvor spezifizierten Aufrufe und die dazu erwarteten Antworten in einem JSON-Format.

Diese Dateien werden anschließend im zweiten Schritt dazu genutzt, die wirkliche Schnittstelle zu testen. Hierzu wird die Schnittstelle tatsächlich mit den spezifizierten Anfragen aufgerufen und die zurückgegebenen Antworten werden gegen die erwarteten geprüft.

Für das Format und den Inhalt der „pact"-Datei gibt es die Pact-Spezifikation [PactSpec]. Dies ist notwendig, da es Pact-Implementierungen für verschiedene Programmiersprachen gibt. Somit kann man die Contracts zum Beispiel mittels JavaScript-Tests definieren und anschließend auf der JVM gegen einen Spring-Boot-Service ausführen.

Pact JVM

Möchte man Pact auf der JVM nutzen, ist Pact JVM [PactJVM] das Mittel der Wahl. Dieses Projekt bietet neben generischen Bibliotheken für Consumer und Provider auch für spezielle Frameworks angepasste Bibliotheken an.

Im Folgenden werde ich anhand eines JAX-RS-Clients als Consumer und einer Spring-Boot-Applikation als Provider zeigen, wie die Verwendung von Pact JVM aussehen kann.

Testen eines Consumers

Für das Beispiel nutzen wir einen in Java, mit Hilfe von JAX-RS, geschriebenen Consumer (s. Listing 1). Um diesen zu testen, bietet sich JUnit und somit der **`pact-jvm-consumer-junit`** [PactJUnitConsumer] an.

```
package de.mvitz.pact.consumer;
import javax.ws.rs.client.*;
import static javax.ws.rs.core.MediaType.TEXT_PLAIN;
public final class Consumer {
  private final WebTarget target;

  public Consumer(WebTarget target) {
    this.target = target;
  }
  public String run() {
    return target.path("/").request(TEXT_PLAIN).get(String.class);
  }
  public static Consumer of(String uri) {
    return new Consumer(ClientBuilder.newClient().target(uri));
  }
  public static void main(String[] args) {
    final Consumer consumer = Consumer.of("http://localhost:8080");
    System.out.println(consumer.run());
  }
}
```

Listing 1: Ein Consumer mittels JAX-RS-Client-API

```
package de.mvitz.pact.consumer;
import au.com.dius.pact.consumer.ConsumerPactTest;
import au.com.dius.pact.consumer.dsl.PactDslWithProvider;
import au.com.dius.pact.model.PactFragment;
import java.io.IOException;
import static org.junit.Assert.assertEquals;

public class ConsumerInheritanceTest extends ConsumerPactTest {
  @Override
  protected PactFragment createFragment(PactDslWithProvider builder) {
    return builder
      .uponReceiving("a root request").method("GET").path("/")
      .willRespondWith().status(200).body("Hello, world!")
      .toFragment();
  }
  @Override
  protected String providerName() {
    return "My Spring Boot Provider";
  }
  @Override
  protected String consumerName() { return "My JAX-RS Consumer"; }

  @Override
  protected void runTest(String url) throws IOException {
    final Consumer consumer = Consumer.of(url);
    assertEquals(consumer.run(), "Hello, world!");
  }
}
```

Listing 2: Ein Consumer-Test mit Vererbung

```
package de.mvitz.pact.consumer;
import au.com.dius.pact.consumer.*;
import au.com.dius.pact.consumer.dsl.PactDslWithProvider;
import au.com.dius.pact.model.PactFragment;
import org.junit.*;
import static org.junit.Assert.assertEquals;

public class ConsumerAnnotationTest {
  @Rule
  public PactProviderRule mockProvider =
    new PactProviderRule("My Spring Boot Provider", this);

  @Pact(consumer = "My JAX-RS Consumer")
  public PactFragment createFragment(PactDslWithProvider builder) {
    return builder
      .uponReceiving("a root request").method("GET").path("/")
      .willRespondWith().status(200).body("Hello, world!")
      .toFragment();
  }
  @Test
  @PactVerification
  public void runTest() throws Exception {
    final Consumer consumer =
      Consumer.of(mockProvider.getConfig().url());
    assertEquals(consumer.run(), "Hello, world!");
  }
}
```

Listing 3: Ein Consumer-Test mit Annotations und Rule

Hierbei haben wir nun die Wahl, ob wir die Tests per Vererbung (s. Listing 2) oder Annotations und Rules (s. Listing 3) schreiben möchten. Beide Ansätze führen zum selben Ergebnis. Pact fährt zu Beginn des Tests einen HTTP-Server hoch und beantwortet die im *Fragment* definierten Aufrufe mit den passenden Antworten. Tätigt unser Consumer andere Aufrufe, merken wir dies schnell durch einen fehlschlagenden Test.

Welche der beiden Varianten man wählt, ist im Endeffekt Geschmackssache. Die Vererbungsvariante erlaubt es uns, lediglich einen Test pro Klasse zu definieren, und funktioniert natürlich nur, solange wir nicht von einer anderen Klasse erben müssen.

Annotations geben mehr Flexibilität. Bei diesen kann man mehrere Tests in einer Klasse definieren, das Erben von einer anderen Klasse ist möglich und es ist auch möglich, einen Consumer gegen mehrere Provider zu testen.

Die „pact"-Datei

Neben dem reinen Testlauf generiert Pact, wie bereits erklärt, eine passende „pact"-Datei (s. Listing 4). Wie erwartet, enthält diese, neben ein paar Metadaten, die definierte Interaktion für den Test.

```
{
  “provider“: { “name“: “My Spring Boot Provider“ },
  “consumer“: { “name“: “My JAX-RS Consumer“ },
  “interactions“: [{
    “providerState“: null,
    “description“: “a root request“,
    “request“: {
      “method“: “GET“,
      “path“: “/“,
      “body“: null
    },
    “response“: {
      “status“: 200,
      “body“: “Hello, world!“
    }
  }],
  “metadata“: {
    “pact-specification“: { “version“: “2.0.0“ },
    “pact-jvm“: { “version“: “3.2.6“ }
  }
}
```

Listing 4: Die generierte „pact“-Datei

Testen des Providers

Zum Testen des Providers wird die vom Consumer generierte „pact“-Datei genutzt. Da die Aufrufe und Antworten dort bereits beschrieben sind, muss lediglich ein wenig Konfiguration ergänzt werden. Neben dem Pfad zu den „pact“-Dateien sind dies in der Regel Protokoll, Host, Port und der Context-Pfad des Services.

Im einfachsten Fall können die „pact“-Dateien direkt vom Build-Tool ausgeführt werden. Pact JVM bietet hierzu für die gängigen Build-Tools Gradle [PactGradle], Leiningen [PactLein], Maven [PactMaven] und SBT [PactSBT] eine Integration an. Alternativ gibt es auch hier wieder eine JUnit-Bibliothek [PactJUnitProvider].

Der mit Spring Boot geschriebene Service (s. Listing 5) kann so zum Beispiel per Maven (s. Listing 6) oder JUnit (s. Listing 7) getestet werden.

```
package de.mvitz.pact.provider;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.*;
import static org.springframework.http.MediaType.TEXT_PLAIN_VALUE;
import static org.springframework.web.bind.annotation.RequestMethod.GET;

@SpringBootApplication
@RestController
public class ServiceProvider {
  public static void main(String[] args) {
    SpringApplication.run(ServiceProvider.class, args);
  }
  @RequestMapping(method=GET, value=“/“, produces=TEXT_PLAIN_VALUE)
  public String index() {
    return “Hello, world!“;
  }
}
```

Listing 5: Beispielhafter Spring-Boot-Service

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <id>pre-integration-test</id>
          <goals>
            <goal>start</goal>
          </goals>
        </execution>
        <execution>
          <id>post-integration-test</id>
          <goals>
            <goal>stop</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>au.com.dius</groupId>
      <artifactId>pact-jvm-provider-maven_2.11</artifactId>
      <version>3.2.6</version>
      <configuration>
        <serviceProviders>
          <serviceProvider>
            <name>My Spring Boot Provider</name>
            <pactFileDirectory>
              src/test/resources/pacts
            </pactFileDirectory>
          </serviceProvider>
        </serviceProviders>
      </configuration>
      <executions>
        <execution>
          <phase>integration-test</phase>
          <goals>
            <goal>verify</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
...
```

Listing 6: Provider-Test mit Maven

```
package de.mvitz.pact.provider;
import au.com.dius.pact.provider.junit.*;
import au.com.dius.pact.provider.junit.loader.PactFolder;
import au.com.dius.pact.provider.junit.target.*;
import org.junit.runner.RunWith;

@RunWith(PactRunner.class)
@Provider(“My Spring Boot Provider“)
@PactFolder(“pacts“)
public class ServiceProviderPactIT {
  @TestTarget
  public final Target target = new HttpTarget(8080);
}
```

Listing 7: Provider-Test mit JUnit

Verteilen von „pact“-Dateien

Nun, da wir gesehen haben, wie man die Tests für die Consumer schreibt und die dort generierten „pact“-Dateien anschließend für die Provider-Tests wiederverwendet, bleibt nur noch die Frage der Verteilung dieser Dateien.

Der einfachste Weg ist, dass der Build-Server die Dateien, sofern sich Änderungen ergeben haben, in das Quellcode-Repository des Providers kopiert und eincheckt. Da dies idealerweise auch wieder zu einem Build des Providers führt, sind die Tests somit bei beiden Projekten immer aktuell und Fehler fallen schnell auf. Zudem sind die Dateien jederzeit lokal verfügbar. Der Nachteil sind die zusätzlichen Commits.

Alternativ können die meisten Provider-Tools nicht nur mit lokalen, sondern auch mit remote erreichbaren „pact“-Dateien umgehen. Der Build des Consumers könnte somit die generierten Dateien einfach per HTTP zugänglich machen. Der Nachteil dieser Alternative ist die zusätzlich benötigte Infrastruktur und der nun benötigte Netzwerkzugriff.

Die dritte Möglichkeit wird von Pact selber bereitgestellt. Mit dem Pact Broker [PactBroker] wird ein Server bereitgestellt, der

neben der reinen Verwaltung der „pact“-Dateien auch noch Zusatznutzen in Form von automatischer Dokumentation, Diagrammen und Versionierung von Pacts anbietet.

Fazit

Gerade in einer Microservices-Architektur bieten sich Consumer-Driven Contracts an, um die intern genutzten Schnittstellen zwischen den einzelnen Services zu definieren und zu überprüfen. Neben dem eigentlichen Contract werden auch Änderungen der Schnittstelle, die nicht rückwärts kompatibel sind, schnell gefunden und können anschließend behoben werden.

Mit Pact gibt es eine Implementierung, die sich auf der JVM nutzen lässt. Die Alternative Pacto [Pacto] fokussiert auf Ruby und ist deshalb für JVM-Projekte nicht die erste Wahl. Die in diesem Artikel gezeigten Listings geben einen ersten Einblick, wie man Pact sowohl für einen Java-Consumer als auch einen Java-Provider nutzen kann.

Links

[CDC] I. Robinson, Consumer-Driven Contracts: A Service Evolution Pattern, 12.6.2006, `http://www.martinfowler.com/articles/consumerDrivenContracts.html`
[ICT] M. Fowler, IntegrationContractTest, 12.1.2011, `http://martinfowler.com/bliki/IntegrationContractTest.html`
[Pact] `http://docs.pact.io`
[PactBroker] `https://github.com/bethesque/pact_broker`
[PactGradle] `https://github.com/DiUS/pact-jvm/blob/master/pact-jvm-provider-gradle`
[PactJUnitConsumer] `https://github.com/DiUS/pact-jvm/tree/master/pact-jvm-consumer-junit`
[PactJUnitProvider] `https://github.com/DiUS/pact-jvm/blob/master/pact-jvm-provider-junit`
[PactJVM] `https://github.com/DiUS/pact-jvm`
[PactLein] `https://github.com/DiUS/pact-jvm/blob/master/pact-jvm-provider-lein`
[PactMaven] `https://github.com/DiUS/pact-jvm/blob/master/pact-jvm-provider-maven`
[Pacto] `http://thoughtworks.github.io/pacto/`
[PactParts] `http://www.pact.io/media/pact_two_parts.png`
[PactSBT] `https://github.com/DiUS/pact-jvm/blob/master/pact-jvm-provider-sbt`
[PactSpec] `https://github.com/pact-foundation/pact-specification`

Michael Vitz ist Consultant bei innoQ und verfügt über mehrjährige Erfahrung in der Entwicklung und im Betrieb von JVM-basierten Systemen. Zurzeit beschäftigt er sich vor allem mit den Themen DevOps, Continuous Delivery und Cloud-Architekturen sowie Clojure.
E-Mail: michael.vitz@innoq.com