Consumer Driven Contract Tests
Vorstellung Hauptprojekt

Alexander Piehl
alexander.piehl@haw-hamburg.de

Hamburg University of Applied Sciences,
Dept. Computer Science,
Berliner Tor 7
20099 Hamburg, Germany

1 Einleitung

In der vorliegenden Ausarbeitung werden die Ergebnisse der im Hauptprojekt stattgefundenen Arbeit präsentiert. Der Schwerpunkt im Hauptprojekt lag darin zu prüfen, inwieweit das Framework PACT nützlich ist zum Testen einer Microservice Architektur. Mit dem Programm PACT können sogenannte Consumer Driven Contract Tests ausgeführt werden. Bei diesen Tests werden die Schnittstellen zwischen zwei Services getestet, die miteinander interagieren [pac17]. Der Testansatz Consumer Driven Contract Tests basiert auf dem Ansatz Consumer Driven Contract, welcher für die Implementierung von Schnittstellen verwendet werden kann. Bei diesem Ansatz werden die Schnittstellen aus der Sicht des Consumers entwickelt. Dabei ist der Consumer der Service, der den anderen Service anfragt, also den Request erstellt. Der angefragte Service wird in dieser Ausarbeitung Provider genannt.

Innerhalb dieser Ausarbeitung soll erarbeitet werden, ob PACT die im Grundprojekt erarbeiteten Anforderungen für das Testen von Microservice Anwendungen erfüllen kann [Pie17]. Die größten Anforderungen waren ein schnelles und aussagekräftiges Feedback aus den Tests zu bekommen, welches ein mehrmaliges Veröffentlichen einer neuen Version pro Tag unterstützt. Dabei dient weiterhin, wie im Grundprojekt, das Framework MARS als Testobjekt.

Da PACT Schnittstellen testet, wird am Anfang der Ausarbeitung der Architekturstil erläutert, mit dem die Schnittstellen von MARS implementiert wurden sind. In MARS sind alle Schnittstellen mit dem Architekturstil REST entworfen worden [CK09]. Nach der grundsätzlichen Erklärung von REST, werden noch Besonderheiten beim Testen definiert, welche sich aus der Schnittstelle und der Verwendung von REST ergeben. Diese neu ausgearbeiteten Herausforderungen sollen die im Grundprojekt ausgearbeiteten Herausforderungen ergänzen. Der Aspekt der Schnittstellen wurde im Grundprojekt zunächst ausgeblendet, da sonst der Umfang zu groß geworden wäre. Im nächsten Kapitel wird anschließend Consumer Driven Contract Test detaillierter vorgestellt und erläutert. Die Erläuterung wird mit einer Beispielimplementierung von PACT unterstützt. Im Anschluss wird die Implementierung von PACT innerhalb von MARS beschrieben, da sich diese vom generellen Ablauf von PACT unterscheidet. Für das

Ausprobieren von PACT innerhalb von MARS wurden zwei Service ausgesucht, an denen beispielhaft die Implementierung ausgeführt wurden ist. Nach der Implementierung der Schnittstellentests mit PACT wurden verschiedene Untersuchungen durchgeführt, die dazu dienten PACT zu bewerten. In dieser Bewertung wird auch eine Aussage getroffen, ob PACT sinnvoll für MARS bzw. generell für eine Microservice Architektur ist. Im letzten Kapitel wird ein Ausblick auf die Masterarbeit gegeben.

2 Grundlagen

2.1 REST

REST ist ein Architekturstil für verteilte Systeme. Die Abkürzung REST steht für Representational State Transfer [CK09]. Erstmals wurde REST im Jahr 2000 von Roy Fielding vorgestellt [KLC13]. Die REST-Architektur wird häufig für Client-Server Anwendungen verwendet, ist jedoch nicht darauf beschränkt. Zurzeit ist REST sehr beliebt bei der Entwicklung von Webservices. Mit REST soll die Implementierung von Webservices leichter sein. Dazu sollen sie auch einfacher zu skalieren sein [CK09]. Auch aus diesen Gründen stellten Google, Facebook und Yahoo ihre Services von SOAP auf REST um [Rod08, NCH+14].

REST basiert dabei auf Resource Oriented Architecture, kurz ROA [CK09]. Dies bedeutet, dass jede wichtige Information als Ressource zur Verfügung stehen muss [PR11]. Die Zugänglichkeit zu den Ressourcen muss über eine Uniform Resource Identifier, kurz URI gegeben sein. Zudem sollen die Ressourcen über verschiedene Methoden manipuliert werden können. Es müssen mindestens die sogenannten CRUD-Operatoren zur Verfügung stehen. CRUD steht für Create, Read, Update und Delete, damit werden die grundsätzlichen Datenoperationen beschrieben. Bei REST werden dafür die im Standard RFC 2616 standardisierten HTTP-Methoden verwendet [KLC13]. In der Tabelle 1 auf Seite 2 werden die Beziehung zwischen den CRUD-Operatoren und den HTTP-Operatoren dargestellt.

CRUD-Operation	HTTP-Methode
Create	POST
Read	GET
Update	PUT
Delete	DELETE

Tabelle 1. Beziehung CRUD und HTTP Operatoren [RVG10]

Neben den CRUD-Operatoren können noch weitere HTTP-Methoden zur Verfügung stehen. Beispiele hierfür sind HEAD und OPTIONS [PR11].

Bei REST müssen die jeweiligen Aufrufe zustandslos erfolgen [RVG10, PR11, KLC13]. Im Detail heißt dies, dass der Webservices keine Informationen über den

Zustand seiner einzelnen Clients speichert. Sollten Informationen über deren Zustand notwendig sein, müssen die Clients die Informationen mit übertragen. Dadurch können auch komplexerer Programmzustände abgebildet werden [PR11].

Die jeweiligen Nachrichten können in verschiedene Formaten vorliegen [RVG10]. Sehr häufig werden XML, JSON oder beide Formate für die Nachrichten verwendet. Eine Vorschrift, welches Format verwendet werden muss, besteht nicht.

Zusammenfassend lässt sich REST in vier Grundprinzipien zusammenfassen [PR11]:

- **Addressability:** Jede wichtige Informationen muss als Ressource vorliegen und über eine eindeutige URI erreichbar sein.
- **Connectedness:** Die Repräsentation ist getrennt von den Ressourcen. Dies bedeutet, Ressourcen können in verschiedenen Formate vorliegen, wie z.B. JSON und XML.
- **Uniform Interface:** Auf die Ressourcen wird nur über standardisierte HTTP-Methoden zugegriffen.
- **Statelessnes:** Jede Kommunikation erfolgt zustandslos.

Webservices, welche die vier Grundprinzipien einhalten, bekommen häufig den Beinamen RESTful [PR11]. Der Begriff RESTful ist nicht eindeutig definiert.

Kontrakte In der allgemeinen Definition bedeutet Kontrakt eine Vereinbarung zwischen zwei unterschiedlichen Parteien. Diese Definition kann sehr gut auf diesen Kontext übertragen werden. Der Kontrakt enthält die Vereinbarungen über die möglichen Interaktionen zwischen dem Service, der anfragt und dem Service, der antwortet [pac17, uHS15].

Verwaltet wird der Kontrakt vom Service, der angefragt werden soll. In dem Sinne ist der Kontrakt eine Beschreibung, welche Funktionen er anderen Services anbietet und wie diese verwendet werden können.

Der Kontrakt ist dabei nichts REST spezielles, sondern wird auch bei anderen Arten von Schnittstellen wie SOAP verwendet. Dabei gibt es kein definiertes Format, in dem der Kontrakt vorliegen soll. Im SOAP-Bereich wird meist eine Datei im WSDL-Format verwendet, während im REST-Bereich viele verschiedene Formate existieren wie z.B. Swagger, RAML und WADL.

Listing 1.1. Beispiel Kontrakt als Swagger-Datei

```
swagger: '2.0'
info:
  description: Service to process uploaded table based imports
  version: '2.0'
  title: Table based importer
host: localhost:5555
basePath: "/"
tags:
- name: tablebased-import-controller
  description: Tablebased Import Controller
```

```
paths:
  "/tablebased/{dataId}":
    delete:
      tags:
      - tablebased-import-controller
      summary: Deletes all persisted data for the given data id.
      operationId: deleteTablebasedFIleUsingDELETE
      consumes:
      - application/json
      produces:
      - "*/*"
      parameters:
      - name: dataId
        in: path
        description: dataId
        required: true
        type: string
      responses:
        '200':
          description: OK
        '204':
          description: No Content
        '401':
          description: Unauthorized
        '403':
          description: Forbidden
```

Das Listing 1.1 enthält einen Beispielkontrakt im Swagger-Format. Wie diesem Beispiel entnommen werden kann, enthält der Kontrakt allgemeine Informationen wie dem Titel des Services und die Adresse, unter dem der Service erreichbar ist.

Dazu enthält der Kontrakt Informationen darüber über welche Pfade die Ressourcen angesprochen werden können und zusätzlich welche HTTP-Methoden zur Verfügung stehen. Ergänzend dazu enthält er auch Informationen, wie die Anfrage aufgebaut werden soll und welche Antworten zu erwarten sind.

Welche und wie die Informationen im Kontrakt abgelegt werden sollen, ist abhängig von dem verwendeten Schnittstellen-Format.

Besonderheiten beim Testen einer REST-Schnittstelle Die Kommunikation zwischen Consumer und Provider erfolgt nach einem festgelegten Kontrakt, welcher auch Schnittstelle genannt wird. Wie im vorherigen Abschnitt bereits erläutert, beschreibt der Kontrakt die Kommunikation zwischen zwei Services. Dahingehend ist es enorm wichtig, dass der Kontrakt eingehalten wird. Andernfalls kann es zu Fehlern führen.

Deswegen muss ein Schwerpunkt beim Testen von Anwendungen, welche REST verwenden, darauf liegen zu kontrollieren, dass der Kontrakt weiterhin

gültig ist. Genau heißt das, dass die angeboten Funktionen so angefragt werden, wie es im Kontrakt definiert ist und dass auch die Antworten im Kontrakt definierten Format vorliegen. Es ist zu vergleichen mit dem Testen eines Methodenaufrufes. Fehlen notwendige Parameter beim Aufrufen oder Werte beim zurückgeben, kann dies zu Fehlern führen. Ähnlich verhält es sich auch beim Testen einer Schnittstelle. Daher muss geprüft werden das Veränderungen am Consumer bzw. Provider nicht zu einer Verletzung des Kontraktes führt.

Besonders bei Anwendungen mit einer Microservice-Architektur, bei der die Kommunikation hauptsächlich über das Netzwerk abläuft, ist es von großer Bedeutung, dass die Kommunikation funktioniert und nicht aufgrund eines fehlerhaften Kontraktes zu Fehlern führt. Wegen der Verlagerung der Kommunikation in das Netzwerk, wird das Überprüfen der Kontrakte umfangreicher, da sehr viele Kontrakte vorliegen. Ergänzend dazu sind die einzelnen Services gleichzeitig Consumer und Provider. Daher kann es unübersichtlich werden, wer welche Services anfragt und wie.

Besonders bei gewünschten Änderungen eines Kontraktes kann es unübersichtlich werden. Aufgrund der möglichen Vielzahl von Consumern, kann der Überblick verloren gehen, welche Consumer aktualisiert werden müssen und welche nicht. Als Beispiel wird das Löschen einer Ressourcen ein weiterer Wert benötigt. In diesem Fall müssen die Consumer ihre Anfrage aktualisieren, damit sie weiterhin die Ressourcen löschen können. Jedoch kann es sein, dass nur ein Teil der Consumer dieser Funktion überhaupt nutzen wollen. Bei einem fehlenden Überblick darüber, welche Consumer welche Anfragen verwenden,kann es zur Folge haben, dass ein Provider Methoden oder Varianten einer Methode bereitstellt, die nicht mehr benötigt werden. Dies kann zu einem unnötig großen Kontrakt führen, der unübersichtlich wird.

Eine andere Problematik, die durch die Verlagerung der Kommunikation in das Netzwerk entsteht ist, dass damit der Traffic im Netzwerk erhöht wird. Dadurch kann es vermehrt auftreten, dass Nachrichten beschädigt werden oder verloren gehen. Mit dieser Problematik müssen die Provider und Consumer umgehen können.

2.2 Consumer Driven Contract Test

Bei dem Ansatz Consumer Driven Contract werden die Schnittstellen bzw. die Kontrakte aus der Sicht des Clients/Consumers definiert [Rob06, uHS15, Vit16]. Dabei spielt es keine Rolle, ob nur einen Consumer oder mehrere existieren. Die jeweiligen Anfragen und gewünschten Antworten werden gesammelt und auf Grundlage dieser Informationen wird der Provider implementiert. Dabei baut der Ansatz auf der Annahme auf, dass der Consumer am besten wüsste, was er nutzen möchte. Dies bedeutet auch, dass zu erst der Kontrakt existiert und erst dann der Provider entwickelt wird.

Wie sich aus der generellen Beschreibung dieses Ansatzes ableiten lässt, eignet er sich am Besten, wenn sowohl die Consumer wie der Provider neu implementiert werden. Bei bestehenden Anwendungen könnte dieses Verfahren unter

anderem dafür genutzt werden, um zu prüfen, ob der Provider unnötige Schnittstellen bzw. Methoden anbietet.

Der Ansatz Consumer Driven Contract hat den Vorteil, dass sehr klar definiert ist welche Anforderungen der Consumer bzw. die Consumer an den Provider haben [Vit16]. Andernfalls muss der Provider ohne Informationen zu den Consumern entscheiden, wie er die jeweiligen Anforderungen umsetzt. Durch die eindeutige Definition der Anforderungen der Consumer an den Provider, kann der Provider so „schlank“ wie möglich implementiert werden. Der Ansatz eignet sich nur für Umgebungen, bei denen Kontrolle über Consumer und Provider besteht [uHS15].

Auf der Basis dieses Konzeptes setzen die Consumer Driven Contract Tests auf [uHS15, Vit16, Rob06]. Ein Framework, welches Consumer Driven Contract Tests unterstützt ist PACT. PACT ist ein Open-Source Tool, welches auf Github verwaltet wird [pac17].

In der Ausführung von Toby Clemson über das Testen von Microservices wird dieses Verfahren explizit empfohlen [Cle14]. Dabei benennt er neben PACT zwei weitere Tools, welche Consumer Driven Contract Tests anbieten. Diese Tools sind PACTO und Janus. Für das Hauptprojekt wurde PACT gewählt, da es nach der ersten Recherche den größten Umfang bietet und zudem besser dokumentiert ist, als die anderen genannten.

Mit PACT können die Consumer Driven Contract Tests in verschiedenen Programmiersprachen geschrieben werden. Zur Auswahl stehen dabei Ruby, C#, JavaScript und Java.

Mit Consumer Driven Contract Tests wird die Einhaltung des Kontraktes getestet und zwar von beiden Seiten. Es wird zu einem getestet, ob der Consumer den Provider korrekt anfragt und zum anderen wird getestet, ob der Provider korrekt antwortet. Das Ziel hierbei ist zu prüfen, dass der Consumer sowie der Provider den Kontrakt einhalten.

Im nächsten Abschnitt wird der generelle Ablauf von PACT anhand eines Beispiels erläutert, bevor im darauffolgenden Abschnitt die Vorteile beschrieben werden, die durch die Verwendung von Consumer Driven Contract Tests entstehen sollen. Denn einige vermeintliche Vorteile leiten sich direkt aus dem Ablauf von Consumer Driven Contract Test ab.

Ablauf Consumer Driven Contract Test In diesem Kapitel wird der generelle Ablauf von Consumer Driven Contract Tests beschrieben. Der Ablauf wird anhand eines Beispiele erläutert [uHS15, Vit16, Vin15]. Dabei findet die Implementierung mit Java statt.

Der Abbildung 1 kann der generelle Ablauf von PACT entnommen werden. Zuerst werden auf der Seite des Consumers die Interaktionen definiert. Dies beinhaltet sowohl die Anfragen wie auch die zu erwarteten Antworten. PACT erstellt auf Grundlage dieser Informationen einen Stub vom Provider, der auf die Anfrage genauso antwortet wie es vorher definiert wurde. Sobald diese Tests erfolgreich abgelaufen sind, erfolgt das Testen des Providers. Dafür werden sozusagen die Seiten gewechselt. Es wird nicht mehr der Provider simuliert, sondern

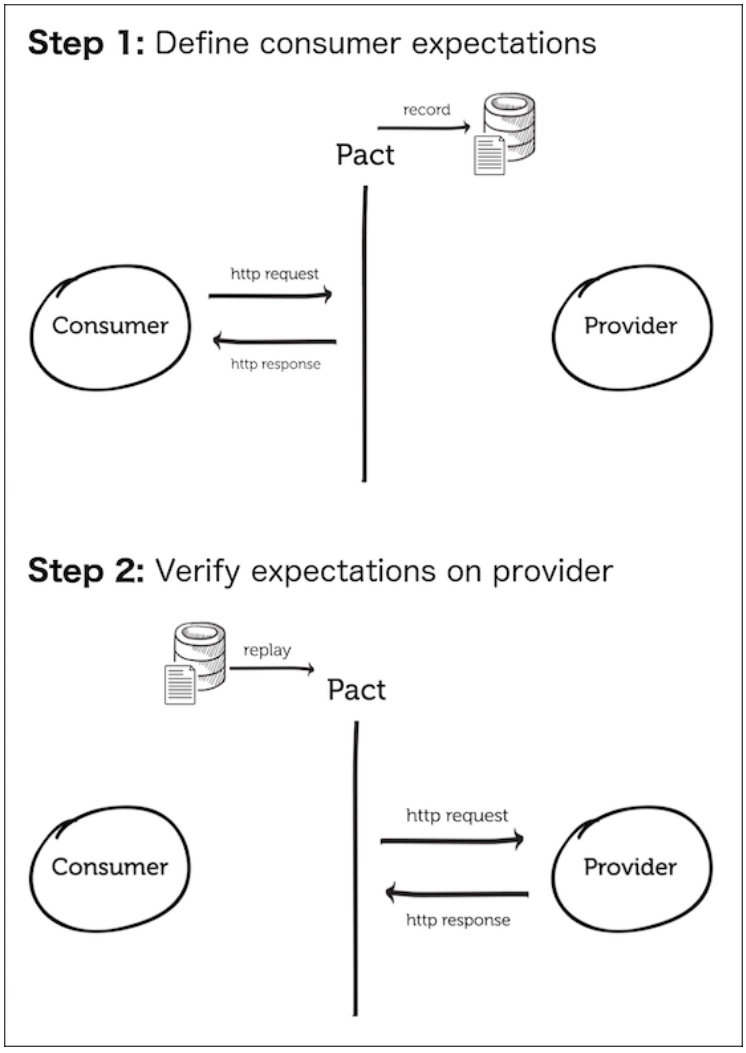

Abb. 1. Ablauf Pact

die einzelnen Consumer. PACT erstellt die Anfrage und überprüft, ob der Provider korrekt antwortet. Damit der Provider getestet werden kann, muss dieser vorher gestartet werden. Beim generellen Ablauf von PACT erkennt man sehr gut die Nähe zu Consumer Driven Contract. Begonnen wird mit den Consumern und abgeschlossen mit dem Provider.

Für die Implementierung von PACT auf der Seite des Consumers wird eine neue Klasse angelegt. Diese Klasse erbt von der PACT Klasse ConsumerPactTest. Durch die Vererbung müssen vier abstrakte Methoden implementiert werden, welche in der folgenden Auflistung genannt werden:

- providerName
- consumerName
- createFragment
- runTest

Mit den Methoden providerName() und consumerName() werden die jeweiligen Bezeichnungen als String zurückgegeben. Sollte man mehrere Test mit unterschiedlichen Consumern und Providern haben, können diese damit besser unterschieden werden und PACT kann die Tests besser zuordnen. Ein Beispiel Implementierung der Methoden ist in Listing 1.2 auf Seite 8 dargestellt.

Listing 1.2. Implementierung von providerName() und consumerName()

```
@Override
protected String providerName() {
    return "Product_Details_Service";
}

@Override
protected String consumerName() {
    return "Product_Service";
}
```

Über die Methode createFragment() werden die Aufrufe des Consumers mit den zu erwartenden Antworten definiert. Dies geschieht mit einer von PACT definierten Domain Specific Language (DSL).

Listing 1.3. Implementierung createFragment()

```
@Override
protected PactFragment createFragment(PactDslWithProvider builder) {
    Map<String, String> headers = new HashMap<>();
    headers.put("Content-Type", "application/json;charset=UTF-8");

    return builder
            .uponReceiving("a request for product details")
            .path("/productdetails/1")
            .method("GET")
            .willRespondWith()
```

```
            .headers(headers)
            .status(200)
            .body("{\"id\":1,\"description\":\
␣␣␣␣␣␣␣␣␣␣␣␣"This is the description for product 1\"}")
            .toFragment();
}
```

Die für das Beispiel implementierte createFragment-Methode ist im Listing 1.3 auf Seite 8 abgebildet. In dieser Methode wird zunächst eine Map erstellt, in der verschiedene Optionen für den Header definiert werden. In diesem Fall wird nur der Content-Type auf JSON mit dem Zeichensatz UTF-8 festgelegt. Als Rückgabewerttyp gibt die Methode ein sogenanntes PactFragment zurück. Dieses PactFragment wird vom PactDslWithProvider erstellt, welches der Methode als Parameter übergeben wird. Daher bekommt es den passenden Namen builder. Mit dem builder können die verschiedenen Interaktionen definiert werden. Die einzelne Eigenschaften einer Interaktionen werden mithilfe von Methoden bestimmt. Die auf Seite 8 im Listing 1.3 verwendeten Methoden werden in der nachfolgenden Auflistung näher erläutert.

- **uponReceiving:** Mit dieser Methode wird eine neue Interaktion erstellt. Als Parameter bekommt sie eine Beschreibung der Interaktion.
- **path:** Der Methode path wird der aufzurufende Pfad als Parameter übergeben.
- **method:** Über die Methode method wird festgelegt mit welcher HTTP-Methode der Aufruf erfolgt.
- **willRespondWith:** Die Methode gibt an, dass ab hier die zu erwartende Antwort definiert wird.
- **headers:** Über dieser Methode kann definiert werden, dass entsprechende Werte im Header vorhanden sein müssen. Es können für Request und Response unterschiedliche Header definiert werden.
- **status:** Hiermit wird der zu erwartende Statuscode definiert.
- **body:** Mit der body Methode wird definiert, welcher Body in der Response erwartet wird.
- **toFragment:** Zum Abschluss wird die Methode toFragment aufgerufen, mit der das Fragment abgeschlossen wird.

Es können auch mehrere Interaktionen mit Hilfe des builders erstellt werden. Jeder Interaktion beginnt mit der Methode uponReciving(). Um eine weitere Interaktion hinzuzufügen, wird einfach nach der Definition der zu erwartenden Antwort mit der Methode uponReceiving() eine neue Interaktion hinzugefügt. Danach erfolgt die Definition der neuen Interaktion nach dem vorher aufgezeigten Schema. In Listing 1.3 wäre dies nach der Methode body().

Sobald das PactFragment, welches die Interaktionen enthält, implementiert ist, kann der Consumergetestet werden. Dafür wird die Methode runTest() umgesetzt, welche auf Seite 9 im Listing 1.4 dargestellt wird.

Listing 1.4. Implementierung runTest()

```
@Override
```

```
protected void runTest(String url) {
    URI productDetailsUri = URI.create
    (String.format("%s/%s/%s", url, "productdetails", 1));

    ProductDetailsFetcher productDetailsFetcher =
    new ProductDetailsFetcher();

    ProductDetails productDetails =
    productDetailsFetcher.fetchDetails(productDetailsUri);

    assertEquals(productDetails.getId(), 1);
}
```

PACT erstellt auf Grundlage des PactFragments einen Stub vom Provider. Die passende URL zum Provider Stub wird der Methode runTest() via Parameter übergeben. Zum einem wird getestet, ob der Consumer eine korrekte Anfrage erstellt hat und zum Anderen ob der Consumer mit der Antwort vom Stub des Providers zurecht kommt. Der Stub vom Provider antwortet auf die Anfrage, wie es im PactFragment definiert wurden ist. Auf die GET-Anfrage antwortet der Provider Stub mit dem Statuscode 200, den definierten Header und Body. Damit wird überprüft, ob der Consumer eine korrekte Antwort vom Provider abarbeiten kann. Dies kann wie in diesem Beispiel mit Asserts gelöst werden.

Die Tests können entweder über ein Plugin zu einem Build Management Tool wie z.B. Maven gestartet werden oder die Klasse kann direkt ausgeführt werden. Ist der PACT Test erfolgreich, wird automatisch das sogenannte PACT-File erstellt. Die für dieses Beispiel erstellte PACT-File ist im Listing 1.5 auf Seite 10 wiedergegeben.

Listing 1.5. PACT-File

```
{
  "provider" : {
    "name" : "Product_Details_Service"
  },
  "consumer" : {
    "name" : "Product_Service"
  },
  "interactions" : [ {
    "description" : "a request for product details",
    "request" : {
      "method" : "GET",
      "path" : "/productdetails/1"
    },
    "response" : {
      "status" : 200,
      "headers" : {
        "Content-Type" : "application/json;charset=UTF-8"
```

```
      },
      "body" : {
        "id" : 1,
        "description" : "This is the description for product 1"
      }
    }
  } ],
  "metadata" : {
    "pact-specification" : {
      "version" : "2.0.0"
    },
    "pact-jvm" : {
      "version" : "2.1.7"
    }
  }
}
```

Das PACT-File ist im Prinzip eine Zusammenfassung der zuvor definierten Klasse. Es enthält die jeweiligen definierten Provider und Consumer Namen, die Interaktionen und die zu erwartenden Antworten. Dazu wird noch angegeben mit welchen Versionen von PACT, die Datei erstellt wurden ist.

Mit dem erstellten PACT-File kann der Provider getestet werden. Hierbei wird getestet, ob der Provider Anfragen, wie sie im Kontrakt definiert sind, verarbeiten kann und vor allem so Antwort, wie es im Kontrakt definiert ist. Dafür muss der Provider entweder das Gradle oder das Maven Plugin von PACT für den Provider verwenden. Um das Plugin verwenden zu können, muss der Pfad zum PACT-File angegeben werden. Die Datei kann sowohl lokal vorliegen oder via einer URL aufgerufen werden. Zusätzlich muss der Pfad des Providers angegeben werden. Die entsprechenden Einstellungen am Beispiel der Verwendung von Gradle sind im Listing 1.6 auf Seite 11 dargestellt.

Listing 1.6. Einstellungen für das Testen vom Provider

```
pact {
    serviceProviders {
    productDetailsServiceProvider {
        protocol = 'http'
        host = 'localhost'
        port = 10100
        path = '/'
        hasPactWith('productServiceConsumer') {
            pactFile =
            file("../Product_Service-Product_Details_Service.json")
            }
        }
    }
}
```

Sobald der Provider gestartet ist, kann das Plugin für den Provider ausgeführt werden. Die Ergebnisse des Tests werden in der Konsole angezeigt.

Vorteile Consumer Driven Contract Test Neben dem generellen Vorteilen, welche man sich von dem Ansatz Consumer Driven Contract Test verspricht und schon im vorherigen Kapitel kurz skizziert wurden, gibt es noch weitere Vorteile, die sich explizit auf das Testen beziehen. Aus Gründen der Übersichtlichkeit wird der Begriff Service wieder verwendet, wenn sich ein Vorteil gleichermaßen auf Consumer und Provider bezieht.

Einer der Gedanken hinter der Microservice-Architektur ist die unabhängige Entwicklung der einzelnen Services. Dementsprechend macht es Sinn auch ein unabhängiges Testen voneinander zu ermöglichen. Beim üblichen Verfahren zum Testen der Zusammenarbeit der verschiedenen Services, also Integrationstests, müssen die einzelnen Services gestartet werden. Dies kann mit ziemlichen Aufwand verbunden sein. Consumer Driven Contract Test verspricht ein unabhängiges Testen der Services dahingehend, ob die Zusammenarbeit mit den anderen Services funktioniert [Vin15, uHS15].

Ein weiterer Vorteil von Consumer Driven Contract Test soll sein, dass jede Änderung am Service dahingehend getestet werden kann, ob sie die definierte Schnittstelle verändert. Damit können Entwickler überprüfen, ob ihre Änderungen zu ungewollten Veränderung der Schnittstellen führen [uHS15].

Dieser Vorteil ist jedoch am Interessantesten bei erwünschten Änderungen am Provider. Mit Consumer Driven Contract Test besteht die Möglichkeit zu evaluieren, ob und welche Consumer angepasst werden müssen, wenn die Schnittstelle des Providers geändert wurde. Besonders bei Anwendungen mit einer Microservice-Architektur kann es unübersichtlich werden, welche Services den Provider anfragen und welche überhaupt aktualisiert werden müssen. Mit Consumer Driven Contract Test soll diese Problematik leichter zu lösen sein [Vit16, Vin15].

Für ein System wie MARS mit einer Microservice-Architektur bei der die Kommunikation via Netzwerk extrem relevant ist, sind Fehler aufgrund von falschen Schnittstellen und/oder falschen Anfragen sehr kritisch. Dazu ändern sich innerhalb von MARS immer wieder die Anforderungen, an die auch die jeweiligen Services angepasst werden müssen. Aufgrund der Architektur besteht die Möglichkeit Änderungen schnell umzusetzen und den Service neu zu deployen. Consumer Driven Contract Test ist dabei sehr vielversprechend diesen Prozess zu begleiten. Es gibt zum einen Überblick über Veränderungen der Schnittstellen, ob gewollt oder ungewollt. Zum Anderen unterstützt es die unabhängige Entwicklung der einzelnen Services. Ob Consumer Driven Contract Test die beschriebenen Vorteile einhält, wird im nächsten Kapitel untersucht, bei dem es um die Umsetzung von Consumer Driven Contract Test innerhalb von MARS geht.

3 Consumer Driven Contract Test in MARS

Nach der grundlegenden Befassung mit PACT soll nun geprüft werden, ob sich PACT für MARS anbietet. Wie schon im Kapitel 3.2 zu den theoretischen Vorteilen von PACT beschrieben, könnte PACT einige Anforderungen an die Tests für MARS erfüllen. Kurz zusammengefasst geht es um ein schnelles Feedback von den Tests, besonders bei Änderungen am Source Code. Ergänzend dazu soll das Feedback aussagekräftig sein, um eine Fehlersuche zu erleichtern.

Im Gegensatz zum eigentlichen gedachten Ablauf von PACT ist MARS ein sehr fortgeschrittenes Projekt, welches kurz vor dem Release steht. Dies hat zur Folge, dass PACT in ein bestehendes System eingebunden werden muss und nicht gleichzeitig mit der Implementierung des Consumers bzw. Providers eingeführt werden konnte.

Ergänzend dazu ist MARS ein sehr umfangreiches System mit vielen verschiedenen Consumern und Providern. Daher wird PACT zunächst nur in einem übersichtlichen Teil von MARS verwendet, um eine Machbarkeitsstudie zu erstellen und im Folgenden zu analysieren ob die Verwendung sinnvoll ist.

Für die Erstellung der Machbarkeitsstudie wurde der Bereich der Imports in MARS ausgewählt. Diese Services haben die Aufgabe Daten und Modelle hochzuladen und damit für die Simulation bereitzustellen. Für diese Entscheidung sprechen mehrere Gründe. Zum Einem ist es für eine Microservice-Architektur ein relatives geschlossenes System und zum Anderem sind die einzelnen Komponenten überschaubar. Dazu ist man in der Lage diesen Bereich des Systems schnell mit Testdaten zu befüllen. Für die Einführung von PACT in MARS werden die Interaktionen zwischen dem Metadata-Client (Consumer) und dem Provider Metadata verwendet.

Wie in Abbildung 2 zu den Services des Imports gut erkenntlich, interagieren alle Services, die für den Import zuständig sind, mit dem Service Metadata. Dieser Service hat die Aufgaben für die einzelne Dateien, die importiert werden, Metadaten zu erstellen, zu aktualisieren und ggf. zu löschen. Die erstellten Metadata werden dann in einer Datenbank abgelegt. Dazu kann das Frontend via des Services Metadata auf die einzelnen Metadata zugreifen. Damit nicht jeder Service, der mit dem Service Metadata interagiert die Aufrufe selber implementieren muss und damit ggf. Redundanzen erzeugt, ist die Kommunikation mit diesem Service im Objekt Metadata-Client gekapselt. Wenn eine Interaktion mit dem Service Metadata erforderlich ist, wird das Objekt Metadata-Client verwendet, welches dann die Interaktionen übernimmt.

3.1 Testumgebung

Die verwendete Testumgebung ist sehr ähnlich zu der im Grundprojekt. Jedoch wurde nicht mehr mit einer virtuellen Maschine gearbeitet, da der VM nicht genug Leistung zugeteilt werden konnte. Daher wurde Linux Ubuntu als Host-System verwendet. Dies bedeutet aber auch, dass kein geschlossenes System mehr für das Testen verwendet werden konnte.

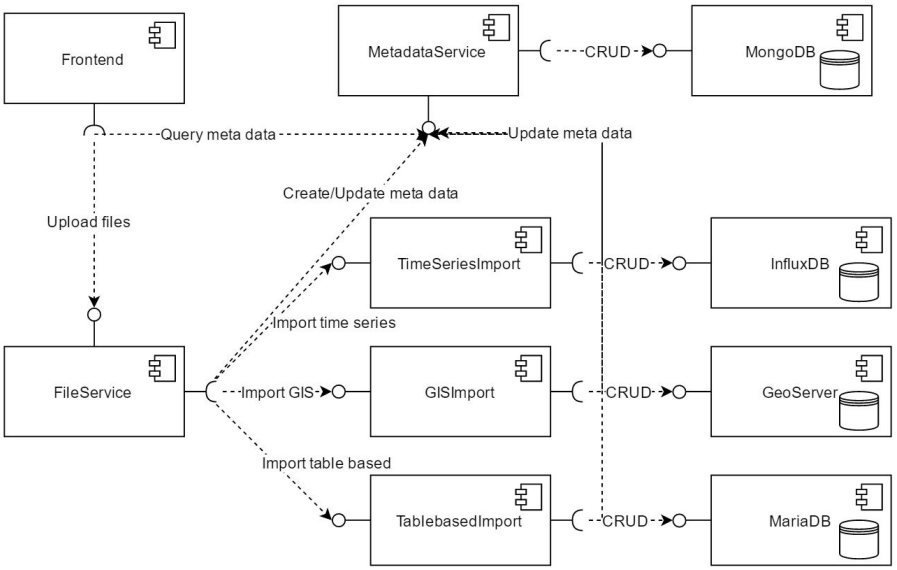

Abb. 2. Diagramm der Komponenten für die Import Services

Zudem wurde wieder Docker in Verbindung mit dem Tool Docker-Compose verwendet, da das Ausführen von MARS ohne Docker nicht möglich ist. Die Installation von MARS erfolgte nach der gleichen Vorgehensweise wie im Grundprojekt.

Ähnlich wie im Grundprojekt wurde auf einem GIT Branch gearbeitet. So wurde von den Import-Projekten ein Branch erstellt. Zum Zeitpunkt der Untersuchung von PACT innerhalb von MARS, gab es stetige Änderungen an den Services für den Import. Dies betraf teilweise auch die Schnittstelle. Um sich ausschließlich auf die Einbindung von PACT zu konzentrieren, wurde der Branch erstellt, damit die Änderungen diesen Prozess nicht beeinträchtigen können.

Für die Implementierung von PACT wurde Intellij als Entwicklungsumgebung verwendet. Dazu wurde Maven als Build-Management-Tool genutzt, da alle MARS-Projekte Maven verwenden und eine Umstellung auf Gradle zu viele Ressourcen verbraucht hätten, ohne im vornherein ermitteln zu können, ob sich dadurch Vorteile ergeben. Ergänzend dazu wurden die Entwicklungswerkzeuge von Chromium und das Programm Postman verwendet. Hiermit konnte man die Interaktionen zwischen den einzelnen Services besser verstehen und MARS konnte mit default Werten gefüllt werden. Zusätzlich wurde das Framework Mockito für die Erstellung von Mocks verwendet.

3.2 Einbindung in MARS

Zwischen dem Metadata-Client und Metadata gibt es mehrere verschiedene Interaktionen. Dabei sind alle CRUD-Methoden, also GET, POST, PUT und DE-

LETE vertreten. Da es zu aufwendig gewesen wäre, wurden nicht alle Interaktionen umgesetzt. In der Tabelle 2 auf Seite 15 sind die Interaktionen beschrieben, welche mit PACT implementiert wurden sind.

Methode	Beschreibung
GET	Bekomme einen Metadata-Eintrag via der spezifischen DataID zurück
GET	Bekomme alle Metadata-Einträge zurück
PUT	Aktualisiere die Variable, die den aktuellen Zustand des Uploads beschreibt
POST	Erstelle einen neuen Metadata-Eintrag
DELETE	Lösche einen Metadata-Eintrag via DataID

Tabelle 2. Interaktionen, die mit PACT umgesetzt wurden sind

Als Soll-Zustand für die Tests wurde der Zustand gewählt, der bei der Erstellung des Branches vorlag. Mithilfe dieses Soll-Zustandes wurden die Interaktionen in PACT implementiert. Als vorbereiteter Schritt davor, mussten die Interaktionen genau analysiert werden, damit klar definiert werden kann, wie die Anfrage und die dazugehörige Antwort auszusehen haben.

Bei MARS werden die APIs mithilfe von Swagger[swa17] dokumentiert. Swagger ist ein open source Framework, welches beim Entwerfen, der Erstellung und der Dokumentation unterstützen soll. Das Framework erstellt für jede API eine extra Datei, in welcher die Schnittstellen beschrieben werden. Der Name der Datei ist im Normalfall swagger.yaml. An der Endung der Datei kann man erkennen, dass sie im YAML-Format vorliegt. In ihr sind alle Methoden beschrieben, die vom Consumer angefragt werden können. Zu den einzelnen Methoden sind wesentliche Informationen angegeben. Unter Anderem der Pfad und die Parameter die erwartet werden. Zu den Parameter ist noch angegeben, welcher Typ vorliegen soll und wie sie versendet werden sollen, also ob sie z.B. im Body oder als Query im Pfad mitgeschickt werden sollen. Die Informationen aus der Swagger-Datei sind im folgenden wichtig für das Programm Postman.

Postman [pos17] ist ein Programm, welches ähnlich wie Swagger bei der Entwicklung und Dokumentation von Schnittstellen unterstützen soll. Allerdings liegt hier der Fokus auf dem Testen von Schnittstellen. Zudem verfügt es im Gegensatz zu Swagger über eine grafische Oberfläche. Mit Postman können Anfragen an einen Provider gestellt werden, um im Anschluss die erhaltenen Antworten zu überprüfen.

Die mit Swagger erstellte Datei kann importiert werden, welche dann auf Grundlage dieser Informationen automatisch Testfälle erstellt. Jedoch fehlt diesen Testfällen Bedingungen, wann sie erfolgreich sind. Diese Bedingungen können mit einer von Postman definierten DSL selbst implementiert werden. Dafür bietet es dafür Templates an, um standardmäßige Überprüfungen auf Status Code oder Ähnliches schnell umsetzen zu können.

Da leider manche Informationen in den Swagger-Dateien von MARS fehlten oder veraltet waren, mussten die automatisch erstellten Tests manuell aktualisiert werden. Dafür waren die Entwicklungswerkzeuge vom Browser Chromium

bzw. Chrome sehr hilfreich. Denn die einzelnen Interaktionen konnten hiermit aufgezeichnet und analysiert werden. Dafür wurden die jeweiligen Interaktionen über das Frontend ausgeführt, aufgezeichnet und anschließend ausgewertet. Mit Hilfe dieser Informationen wurden die Testfälle innerhalb von Postman aktualisiert. Sobald alle Tests implementiert waren und erfolgreich abliefen, wurde damit begonnen PACT auf der Seite des Consumers einzubinden. Der Schritt über Postman war notwendig, um die die Interaktionen besser verstehen zu können. Erst dadurch war es möglich die Interaktionen in PACT umzusetzen. Jedoch gab es ein Problem mit der DELETE-Methode. Beim Testen konnte der Aufruf nicht ausgeführt werden. Der Service antwortete mit dem Statuscode 405. Dies bedeutet, dass die gewählte Methode nicht erlaubt ist. Die Interaktion wurde jedoch in PACT implementiert, da es sich um einen Fehler in MARS handelte und der Service die Anfrage nicht korrekt verarbeitete. Dahingehend wurde die Implementierung der Interaktion mit der DELETE-Methode in PACT alleine auf Grundlage der Swagger-Datei ausgeführt.

Implementierung des Consumers Bei der Implementierung von PACT beim Consumer wurde ähnlich vorgegangen, wie im allgemeinen Beispiel beschrieben. Hierbei wird getestet, ob der Consumer die Anfragen so stellt, wie es im Kontrakt definiert ist. Auch hier wurde zunächst eine neue Klasse erstellt, die Klasse MetadataClientConsumerTest, welche von ConsumerPactTest erbt. Damit waren wieder die vier Methoden providerName(), consumerName(), createFragment() und runTest() vorgegeben. Der Consumer bekam den Namen metadata-client und der Provider den Namen metadata-service.

In den Methoden createFragment und runTest sind jeweils alle fünf Interaktionen abgebildet. Aus Gründen der Übersichtlichkeit wird die Umsetzung einer Interaktionen nach der Anderen beschrieben.

Begonnen wurde mit der Interaktionen, bei der via GET ein spezifischer Metadata Eintrag abgefragt werden soll. Der für diese Interaktion spezifische Code-Abschnitt in der Methode createFragment ist im Listing 1.7 abgebildet.

Listing 1.7. createFragment für die GET-Methode

```
.uponReceiving("Get a metadata entry")
.path("/metadata/" + dataID)
.method("GET")
.headers(headerReq)
.query("")
.body("")
.willRespondWith()
.headers(headersRes)
.status(200)
.body(createJsonBody())
```

Wie in der Beschreibung von REST erwähnt, muss jede Ressource über eine eindeutige URI erreicht werden können. Bei den Metadata Einträgen erfolgt dies

über die Eigenschaft DataID. Daher wird dem Pfad der Wert der DataID angehängt. Weitere Informationen sind in der Query und im Body nicht notwendig. Im Header ist definiert wurden, dass nur JSON akzeptiert wird. Der Provider soll auf diesen Aufruf mit dem Statuscode 200 antworten, wie es im Kontrakt definiert ist. Obwohl ein Header für die Antwort definiert ist, findet herauf keine Überprüfung statt, da der Header keine Werte enthält.

Bis auf die einzelnen Werte unterscheidet sich dieses PactFragment nicht vom generellen Beispiel. Nur der Body, welcher die Antwort enthalten soll, ist deutlich umfangreicher. Dafür wird beim Body die Methode createJsonBody() aufgerufen, die den in der Antwort enthaltenen Body erstellt. Die Daten im Body der Antwort sollen im JSON-Format vorliegen. Diese Methode ist abgebildet im Listing 1.8 auf Seite 17. Ähnlich wie bei der Erstellung des PactFragments wird auch für die Erstellung des Body eine von PACT definierte DSL genutzt.

Listing 1.8. Erstellung eines JSON-Body

```
private PactDslJsonBody createJsonBody() {
    PactDslJsonBody body = new PactDslJsonBody()
        .stringValue("dataId",dataID)
        .stringType("title")
        .stringType("description")
        .integerType("projectId")
        .integerType("userId")
        .stringValue("privacy","PRIVATE")
        .stringValue("state","FINISHED")
        .stringValue("errorMessage",null)
        .stringValue("records",null)
        .stringValue("type",null)
        .stringValue("geoindex",null)
        .object("additionalTypeSpecificData")
            .object("topLeftBound")
                .decimalType("lng")
                .decimalType("lat")
            .closeObject()
            .object("bottomRightBound")
                .decimalType("lng")
                .decimalType("lat")
            .closeObject()
        .closeObject()
        .asBody();
    return body;
}
```

Wie man aus Listing 1.8 gut ableiten kann, ist das Metadata-Objekt recht umfangreich. Da bei der Anfrage definiert wurde, dass nur JSON akzeptiert wird, muss die Antwort im JSON-Format vorliegen. Um komplexere Antworten in JSON zu definieren, bietet PACT unter anderem die Klasse PactDslJsonBody

an. Damit kann ein Objekt erstellt werden, welches unterschiedliche Eigenschaften haben kann.

Generell wird der Typ der jeweiligen Eigenschaften bestimmt. Wenn kein Wert angegeben wird, wird ein zufälliger Wert vom festgelegten Typ erstellt. Dazu muss immer der Name der jeweiligen Eigenschaft mit angegeben werden. Die bei der Erstellung des JSON-Body verwendeten Methoden sind in der Tabelle 3 auf Seite 18 näher beschrieben.

Methode	Beschreibung
stringValue	Name, Typ und Wert müssen übereinstimmen.
stringType	Name und Typ müssen übereinstimmen.
intergerType	Name und Typ müssen übereinstimmen.
decimalType	Name und Typ müssen übereinstimmen.
object	Erzeugung eines Objektes. Name muss übereinstimmen
closeObject	Schließen des Objektes
asBody	Abschließende Methode, um in den Typ PactDslJsonBody zu casten.

Tabelle 3. Erklärung der verwendeten PACT DSL Methoden

Werte werden nur überprüft, wenn sie explizit mit einer der Methoden festgelegt wurden sind. In dieser Interaktion wird z.B. der Wert der DataID überprüft, da dieser identisch zu dem bei der Anfrage verschickten Wert sein muss. Soll darauf geprüft werden, dass eine Eigenschaft keinen oder den Null-Wert enthält, dann muss explizit der Null-Wert übergeben werden. Andernfalls wird immer ein Zufallswert für die Eigenschaft erstellt. Die Überprüfung erfolgt im Hintergrund automatisch über sogenannte Matcher. Die Matcher werden zeitgleich mit der Erstellung der Eigenschaften erstellt. Dabei prüfen die Machter entweder, ob der Typ korrekt ist oder ob Typ und Wert korrekt sind. Es kommt darauf an, wie die Eigenschaften angelegt wurde.

Nun musste für diese Interaktion noch die Methode runTest() implementiert werden. Der dazu passende Code-Abschnitt ist im Listing 1.9 auf Seite 18 abgebildet.

Listing 1.9. runTest() für die GET-Methode

```
EurekaClientMock mockEureka = new EurekaClientMock();
MetadataClient client = MetadataClient
.getInstance(new RestTemplate(), mockEureka.getMockEurekaClient());

URI metadataEntryUri = URI
.create(String.format("%s/%s/%s", url, "metadata", dataID));

Metadata metadata = client.fetchMetaData(metadataEntryUri);
assertEquals(dataID, metadata.getDataId());
```

Wie schon in der Testumgebung beschrieben, wird das Framework Mockito verwendet. Die Verwendung ist notwendig um den EurekaClient zu mocken. Eu-

reka [Ran12] ist ein von Netflix entwickeltes Programm für die Service-Discovery. Vereinfacht ausgedrückt verwaltet es die einzelnen Adressen der Services. Dieser Service steht beim Testen des Consumers nicht zur Verfügung, da nur der Consumer gestartet werden soll und keine weiteren Services. Jedoch wird für die Erstellung eines Objektes von Metadata-Client ein EurekaClient-Objekt benötigt. Die eigentliche Aufgabe und zwar die Mitteilung der Adresse des gesuchten Services, erfüllt der Mock nicht. Denn PACT erstellt zur Laufzeit eine Adresse für den gestubbten Provider. Bisher war es nicht möglich diese Information in den gemockten EurekaClient unterzubringen. Daher wurde der MetadataClient um Methoden erweitert, die nicht den EurekaClient für die Adresse des Service anfragen, sondern die Adresse als Parameter übergeben bekommen. Eine Auswirkung sollte diese Änderung nicht haben, da ausschließlich die Bereitstellung der Service Adresse geändert wurde. Diese Umstellung betrifft alle Interaktionen.

Nachdem die URI zusammengebaut wurden ist, kann diese URI dem Client übergeben werden. Dafür wird die Methode fetchMetaData() aufgerufen, welche die Anfrage startet und den Metadata-Eintrag zurückliefert. Ist die Anfrage korrekt, wie sie im PactFragment definiert wurden ist, schickt der Stub vom Provider die im PactFragment definierte Antwort zurück. Zur Überprüfung der regelgerechten Verarbeitung wird der Wert der DataID überprüft, welche identisch mit dem Wert der DataID bei der Anfrage sein soll. Dies wird mit einem Assert überprüft.

Damit ist die erste Interaktion implementiert. Die zweite Interaktion ist auch eine GET-Methode, bei der alle Metadata-Einträge abgefragt werden. Die Implementierung dieser Interaktion unterscheidet sich kaum von der eben gerade beschriebenen Interaktion. Nur der Body in der Response unterscheidet sich. Es wird nun ein Array von Metadata-Objekten erwartet. Die für die Erstellung dieses Array zuständige Methode ist verkürzt im Listing 1.10 auf Seite 19 dargestellt.

Listing 1.10. Erstellung eines Array

```
return PactDslJsonArray
    .arrayEachLike()
        ...
    .closeObject();
```

Mit der Methode arrayEachLike() wird sichergestellt, dass jedes Objekt in der Liste dem hier definierten Beispiel gleichen soll. Ab dem Aufruf der Methode arrayEachLike wird das Objekt definiert. Auch diese Methode muss mit einem closeObject() abgeschlossen werden. Es wird das selbe Objekt definiert, wie es im Listing 1.8 auf Seite 17 dargstellt ist.

Auch in der runTest-Methode gibt es keine großen Unterschiede. Es wird eine andere Methode vom Metadata-Client aufgerufen, als die im Listing 1.9 dargestellte Methode, da eine Liste mit Metadata-Objekten erwartet wird. Ergänzend hierzu ist die assert-Methode angepasst.

Als nächstes wird die PUT-Interaktion implementiert. Diese unterscheidet sich von den bisherigen Implementierungen. Neben den Unterscheidungen im

Header der Anfrage, muss für diese Interaktion eine Query definiert werden. Denn die Änderung des Upload-Zustandes erfolgt über eine Query in der URI. Dazu muss für diese Interaktion kein Body definiert werden , da die Antwort kein Body liefert. Es wird ausschließlich über den Statuscode abgebildet, ob die Anfrage erfolgreich war. Die Implementierung in der runTest-Methode unterscheidet sich nur im Aufruf einer anderen Methode, die die PUT-Anfrage stellt und in der Überprüfung des Ergebnisses.

Als nächstes wurde die Interaktion mit der POST-Methode umgesetzt. Mit dieser Interaktion soll ein neuer Metadata-Eintrag erzeugt werden. Das Erstellen des Fragements unterscheidet sich daher stärker von den bisherigen Interaktionen. Dieses Mal muss der Body für die Anfrage definiert werden. Denn mit der Anfrage müssen die einzelnen Informationen für den Eintrag mitgeschickt werden. Bei der Antwort wird außerdem der Statuscode 201 erwartet und nicht wie bisher der Statuscode 200. Der Statuscode 201 bedeutet, dass ein neuer Eintrag erfolgreich erstellt wurden ist. Im Body der Antwort wird eine UUID als Plain-Text erwartet. Diese UUID wird als DataID verwendet.

Zum Ausführen der POST-Interaktionen wird in der runTest-Methode die Methode für die POST-Methode aufgerufen, welche die einzelnen Informationen zum Metadata-Eintrag als Parameter übergeben bekommt. Da der genaue Wert der UUID nicht vorhergesagt werden kann, wird kein Assert auf die UUID übernommen. Es wird nur geprüft, ob eine UUID zurück geliefert wird. Diese Überprüfung übernimmt PACT.

Die Implementierung der DELETE-Interaktion ist ähnlich wie die erste GET-Interaktion. Im Gegensatz zu dieser wird die DELETE-Methode verwendet. Zusätzlich kein Body in der Antwort erwartet.

Der Aufruf in der runTest-Methode unterscheidet sich in der Methode, die vom Metadata-Client aufgerufen wird. Da kein Body zurück geliefert wird, findet nur eine Überprüfung auf den Wert des Statuscodes statt. Auch diese Aufgabe übernimmt PACT.

Hiermit ist die Implementierung auf der Seite des Consumers abgeschlossen und es können die PACT-Tests auf dieser Seite ausgeführt und das PACT-File erstellt werden. Im Anschluss kann PACT auf Seite des Providers eingebunden werden.

Implementierung Provider Die Einbindung von PACT auf Seiten des Providers ist deutlich übersichtlicher als auf der Seite des Consumers. Beim Provider muss, wie oben in der allgemeinen Beschreibung schon vorgestellt, nur das Plugin eingebunden werden. Da MARS Maven verwendet, wurde das Maven-Plugin verwendet. Im Listing 1.11 auf Seite 20 ist der Abschnitt der pom.xml vom Metadata-Service abgebildet, die für das PACT-Plugin benötigt wird.

Listing 1.11. Eintrag in der pom.xml

```
<plugin>
    <groupId>au.com.dius</groupId>
    <artifactId>pact-jvm-provider-maven_2.11</artifactId>
```

```
    <version>3.3.3</version>
    <configuration>
        <serviceProviders>
            <serviceProvider>
                <name>metadata-service</name>
                <protocol>http</protocol>
                <host>localhost</host>
                <port>80</port>
                <path>/metadata</path>
                <consumers>
                    <consumer>
                        <name>metadata-client</name>
                        <pactFile>
                        ../metadata-client-metadata-service.json
                        </pactFile>
                    </consumer>
                </consumers>
            </serviceProvider>
        </serviceProviders>
    </configuration>
</plugin>
```

In diesem Ausschnitt aus der pom.xml findet sich unter anderem die Namen vom Provider und Consumer wieder. Zudem muss die Adresse angeben werden, unter der der gestartete Service gefunden werden kann. Wie schon in der allgemeinen Beschreibung von PACT erwähnt, kann PACT den Provider nur testen, wenn er ausgeführt wird. Dazu befindet sich der Pfad zum PACT-File in diesem Ausschnitt, welcher aus Gründen der Übersicht nur verkürzt dargestellt ist. Mit der Einbindung des Plugins kann der Verify-Job vom PACT-JVM-Provider-Plugin ausgeführt werden. Mit der Ausführung dieses Jobs werden die Tests auf der Seite des Providers ausgeführt.

3.3 Untersuchung

Mit der Einbindung von PACT konnten verschiedene Untersuchungen durchgeführt werden, um zu prüfen ob die Verwendung von PACT innerhalb von MARS bzw. einer Microservice Architektur sinnvoll ist. Dafür mussten einzelne Testfälle definiert werden, welche alltägliche Situationen widerspiegeln. Dabei werden in diesem Kapitel die Testfälle beschrieben und die jeweiligen Ergebnisse skizziert. Die Einordnung und Bewertung der Ergebnisse findet im nächsten Kapitel statt. Auf der Seite des Consumers wurden zwei Testfälle und beim Provider vier Testfälle definiert.

Beim ersten Testfall verarbeitet der Consumer das Ergebnis vom PACT erstellten Provider nicht korrekt. Über die GET-Anfrage, mit Angabe einer spezifischen DataID, gibt PACT den vorher definierten Metadata-Eintrag zurück. Für Testzwecke wurde auf Seiten des Consumers das Metadata-Objekt dahingehend

angepasst, dass es immer die DataID „0“ zurück gibt. Damit konnte geprüft werden, wie PACT mit der Rückgabe einer falschen DataID umgeht.

Da in der runTest-Methode ein Assert auf die DataID implementiert wurden ist, schlägt dieser fehl. Die dazugehörige Meldung ist in Abbildung 3 auf Seite 22 abgebildet. Dieser Fehler wurde jedoch nur aufgrund des vorher definierten Asserts gefunden. Sollten weitere Fehler im Metadata-Objekt vorhanden sein, werden diese nicht gefunden.

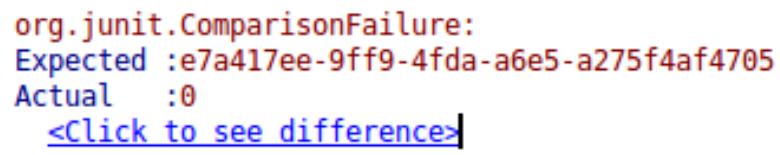

Abb. 3. Fehlgeschlagenes Assert aufgrund der falschen DataID

Beim zweiten Testfall auf der Seite des Consumers wurde mit Absicht eine Eigenschaft des Headers im Request falsch gesetzt. Diese Art von Fehler kann auftreten, wenn bei der Implementierung des Aufrufes der Header falsch gesetzt wird, aber bei der Implementierung des PACT-Fragments korrekt. Beim POST-Aufruf soll der Content-Type „application/json“ sein. Dies ist auch so im PACT-Fragment definiert. Hier wurde jedoch der Aufruf angepasst, so dass der Content-Type auf „text/plain“ festgelegt wurde. In der Konsole erschien darauf die im Listing 1.12 auf 22 dargestellte Meldung, wobei der Stacktrace gekürzt wurde.

Listing 1.12. Fehler im Header

```
org.springframework.web.client.HttpServerErrorException:
500 Internal Server Error
..

au.com.dius.pact.consumer.PactMismatchException:
Pact verification failed for the following reasons:
createMetadata:
    HeaderMismatch - Expected header 'Content-Type'
    to have value 'application/json' but was 'text/plain'
    BodyTypeMismatch(application/json,text/plain)

The following requests were not received:
Interaction: createMetadata
        in state None
request:
        method: POST
        path: /metadata
```

```
        query: null
        headers: [Content-Type:application/json]
        matchers:
        [$.body.projectId:[match:integer], $.body.userId:[match:integer]]
        body: au.com.dius.pact.model.OptionalBody(PRESENT,
        {"description":"Test Metadate created via PACT","privacy":"PRIVATE",
        "title":"Test Metadata PACT","type":"GIS","projectId":1,"userId":1})

response:
        status: 201
        headers: [Content-Type:text/plain]
        matchers: [$.body:
        [regex:[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}]]
        body: au.com.dius.pact.model
        .OptionalBody(PRESENT, ba501910-5b3c-4bae-ac68-a56bafc793f2)
```

Beim ersten Test für den Provider sollte dieser einen fehlerhaften Body zurückgeben. Dafür wurde die GET-Interaktion verwendet, welche einen spezifischen Metadata-Eintrag zurück gibt. Da zum Zeitpunkt des Testens der Geoserver, welcher bei der Erstellung von Metadata-Einträgen aufgerufen wird ausgefallen war, waren alle erstellten Testdaten defekt. Dies kam bei diesem Testfall sehr entgegen, da keine Daten manuell manipuliert werden mussten. Der Testfall ist dahingehend besonders interessant, da dies wahrscheinlich ein häufiger Fehler ist, der beim Testen von Schnittstellen auftauchen kann. Wie in vorherigen Kapiteln beschrieben, können Änderungen am Provider die Schnittstelle verändern. Dadurch kann ein Objekt anders zurückgeben werden als es erwartet wurden ist.

Durch den Ausfall des Geoservers enthielten alle Metadata-Einträge keine Geo-Informationen, welche in der MAP „additionalTypeSpecificData“ gespeichert werden. Dieser Map fehlten also alle Metadata-Einträge. Das Fehlen produzierte die auf Seite 23 im Listing 1.13 dargestellte Fehlermeldung.

Listing 1.13. Test

```
Verifying a pact between metadata-client and metadata-service
  [Using file /home/alex/Mars/mars-import/metadata-client/
  target/pacts/metadata-client-metadata-service.json]
  Get a metadata entry
    returns a response which
      has status code 200 (OK)
      includes headers
        "Content-Type" with value "application/json; charset=UTF-8" (OK)
      has a matching body (FAILED)

Failures:

0) Verifying a pact between metadata-client and metadata-service -
```

```
Get a metadata entry returns a response which has a matching body
      $.body.additionalTypeSpecificData -> Type mismatch:
      Expected Map Map(bottomRightBound ->
      Map(lat -> 1954648900, lng -> 1613291745),
      topLeftBound -> Map(lat -> 4134225878, lng -> 3873370324))
      but received Null null

      Diff:

      @1
      -    "additionalTypeSpecificData": {
      -        "bottomRightBound": {
      -            "lat": 1954648900,
      -            "lng": 1613291745
      -        },
      -        "topLeftBound": {
      -            "lat": 4134225878,
      -            "lng": 3873370324
      -        }
      -    },
      +    "additionalTypeSpecificData": null,
           "dataId": "86fc448c-7cac-48cf-88e1-e3bbdfc2452e",

      @12
           "dataId": "86fc448c-7cac-48cf-88e1-e3bbdfc2452e",
      -    "description": "CdWPkGgzoyOIPJJbjZah",
      +    "description": "GIS Import via Postman",
           "errorMessage": null,

      @16
           "privacy": "PRIVATE",
      -    "projectId": 376674471,
      +    "projectId": 1,
           "records": null,

      @19
           "state": "UPLOADED",
      -    "title": "NPQsfuCfIeUsNkKmnczm",
      +    "tags": null,
      +    "title": "Postman Import",
           "type": "GIS",

      @21
           "type": "GIS",
      -    "userId": 994762534
```

```
+     "userId": 1
}
```

Im zweiten Testfall für den Provider wurde erneut der Header manipuliert. Dieses mal auf der Seite der Response. Für diesen Testfall wurde die PUT-Interaktion ausgewählt, die den Wert aktualisiert, welcher den aktuellen Zustand des Uploads einer Datei beschreibt. Dafür wurde der Fehler im PACT-Fragment gesetzt, anstatt den Metadata-Service anzupassen, da dies deutlich weniger Aufwand erforderte. In dem erwarteten Response-Header wurde die Eigenschaft „Content-Length“ auf „1“ gesetzt. In der Antwort vom Povider ist der Wert im Header „0“, da der Response Body auf die PUT-Abfrage leer ist. Dies führte zu der auf Seite 25 im Listing 1.14 aufgeführten Meldung.

Listing 1.14. Test

```
Verifying a pact between metadata-client and metadata-service
  [Using file ../metadata-client-metadata-service.json]
  Set a new state for a metadata entry
    returns a response which
      has status code 200 (OK)
      includes headers
        "Content-Length" with value "1" (FAILED)
      has a matching body (OK)

Failures:

0) Verifying a pact between metadata-client and metadata-service
- Set a new state for a metadata entry returns
a response which Verifying a pact between metadata-client
and metadata-service
  [Using file ../metadata-client-metadata-service.json]
  Set a new state for a metadata entry
    returns a response which
      has status code 200 (OK)
      includes headers
        "Content-Length" with value "1" (FAILED)
      has a matching body (OK)

Failures:

0) Verifying a pact between metadata-client and metadata-service
- Set a new state for a metadata entry returns
a response which includes headers "Content-Length" with value "1"
      Expected header 'Content-Length' to have value '1' but was '0'
```

Beim dritten Testfall soll der Statuscode nicht mit dem vorher definierten Statuscode übereinstimmen. Dieser Fehler kann leicht auf der Seite des Providers auftauchen, wenn der Provider mit einer Anfrage nicht umgehen kann

oder aufgrund von „Copy and Paste“ ein falscher Statuscode für die Rückgabe definiert wurde.

Der Testfall konnte leicht umgesetzt werden, indem mit der DELETE-Interaktion ein Metadata-Eintrag gelöscht werden soll, welcher nicht vorhanden ist. Daher gibt der Provider den Statuscode 404 zurück und nicht wie im PACT-Fragment definiert 200. Dies führte zu der im Listing 1.15 auf Seite 26 abgebildeten Meldung.

Listing 1.15. Test

```
Verifying a pact between metadata-client and metadata-service
  [Using file ../metadata-client-metadata-service.json]
  Delete a metadata entry
    returns a response which
      has status code 200 (FAILED)
      has a matching body (OK)

Failures:

0) Verifying a pact between metadata-client and metadata-service
- Delete a metadata entry returns a response which has status code 200
      assert expectedStatus == actualStatus
             |              |  |
             200            |  404
                            false
```

Der letzte Testfall war im Gegensatz zu den vorherigen Fällen nicht von Anfang an geplant. Beim Erstellen eines neuen Metadata-Eintrags mithilfe der POST-Interaktion soll eine DataID, welche im UUID-Format vorliegen soll, als einfacher Text zurückgegeben werden. Dafür wurde im PACT-Fragment definiert, dass die Antwort eine UUID ist und der Content-Type „plain/text“ ist. Die DataID wird für jeden neuen Metadata-Eintrag neu erstellt und der Wert wird zufällig bestimmt. Deswegen ist im PACT-Fragment die Überprüfung auf den Response Body so definiert wurden, dass keine exakte Überprüfung des Wertes stattfindet, sondern eine Überprüfung über reguläre Ausdrücke. Mit der Überprüfung via reguläre Ausdrücke wird getestet, ob der zurückgegebene Wert im UUID-Format vorliegt. Obwohl das Fragment für diese Überprüfung so erstellt wurde, wie es in der PACT Dokumentation beschrieben wurden ist, erschien die auf Seite 26 in Listing 1.16 abgebildete Fehlermeldung. Wie man der Meldung entnehmen kann, schlug die Überprüfung fehl, da die DataIDs unterschiedlich sind. In der Dokumentation gab es keine weitergehenden Informationen zu diesem Verhalten.

Listing 1.16. Fehler in PACT

```
Verifying a pact between metadata-client and metadata-service
  [Using file ../metadata-client-metadata-service.json]
  createMetadata
    returns a response which
```

```
    has status code 201 (OK)
    includes headers
      "Content-Type" with value "text/plain" (OK)
    has a matching body (FAILED)

Failures:

0) Verifying a pact between metadata-client and metadata-service
- createMetadata returns a response which has a matching body
    / -> mismatch

    Diff:

    @0
    -d08f46fd-95e9-432c-95c2-7b1a0a078162
    +326e27d7-6285-45de-8343-36b3ab3798f8
```

Damit sind alle Testfälle beschrieben, welche ausgeführten wurden, um PACT bewerten zu können.

3.4 Bewertung der Ergebnisse

Das Fazit und die damit verbundene Bewertung für PACT soll in zwei grobe Bereiche unterteilt werden. Zum Einen die Einbindung von PACT in ein bestehendes System und zum Anderen die Bewertung der ausgeführten Testfälle.

Im Gegensatz zu dem am Anfang beschriebenen Ansatz, bei dem PACT bei der Implementierung des Consumers mit eingebunden wird, musste bei MARS PACT in ein bestehendes System eingebunden werden. Ergänzend hierzu kam, dass sowohl der Metadata-Client als auch der Metadata-Service von Anderen implementiert wurde. Im generellen Ablauf von PACT wird jedoch davon ausgegangen, dass Consumer und Provider von den selben Personen implementiert wurde, die auch PACT implementieren (wollen). Zumindest werden sehr genaue Kenntnisse vorausgesetzt. Dieser Umstand führte bei dieser Arbeit zu einigen Schwierigkeiten. Denn es waren nicht immer alle notwendigen Informationen über die Interaktionen vorhanden. Zwar lieferten die Swagger-Dateien wichtige Informationen, jedoch waren diese nicht immer korrekt oder veraltet. Zudem fehlten auch einige Informationen in der Swagger-Dateien. Auch wenn man diese Informationen teilweise über Analyse der Aufrufe via Chrome herausarbeiten konnte, war dies jedoch mit hohem Aufwand verbunden. Dieser Mehraufwand wäre vermutlich nicht vorhanden, wenn man das System genau kennen würde.

Aufgrund der fehlenden Informationen konnte auch nicht immer entschieden werden, ob das aktuelle Verhalten korrekt ist. Deswegen gab es teilweise auch keine andere Möglichkeit als das aktuelle Verhalten als das Soll-Verhalten zu definieren. Zudem konnte wegen den fehlenden Informationen keine Priorisierung vorgenommen werden. Es war nicht klar, welche Aspekte unbedingt getestet werden müssen und welche nicht. Ein einfaches Beispiel hierfür ist, dass teilweise

nicht festgestellt werden konnte, ob eine bestimmte Eigenschaft des Headers getestet werden muss oder nicht.

Um die einzelnen Interaktionen besser zu verstehen, war es unumgänglich diese zuerst in Postman umzusetzen. Denn hiermit ergab sich eine bessere Übersicht, wie genau die Anfragen auszusehen haben und wie die Antworten aufgebaut sind. Erst mit diesen Informationen konnte die Implementierung von PACT beginnen.

Jedoch erfolgte die Implementierung von PACT nicht ganz ohne Schwierigkeiten. Das generelle Beispiel und die danach erfolgte Implementierung in MARS sieht auf den ersten Blick sehr kompakt und vor allem simpel aus. Im Prinzip ist es das auch, sobald das Verständnis vorliegt. Jedoch ist die Dokumentation von PACT ausbaufähig, was das Aufbauen eines benötigten Verständnisses von PACT erschwerte.

In der PACT Dokumentation waren viele Aspekte nur kurz erläutert. Teilweise waren Aspekte nur mithilfe eines Beispieles ohne aussagekräftige Beschreibung vorgestellt. Problematisch wurde es, wenn die gezeigten Beispiele nicht für die aktuelle Situation übertragen werden konnten. Dann konnte nur die teilweise sehr kurze Erklärung zu Rate gezogen werden. Dies führte teilweise dazu, dass einige Abschnitte mit der Methode „trial and error“ implementiert werden mussten, also so lange ausprobierte, bis es funktionierte. Besonders die erste Erstellung eines JSON-Body mit PACT war sehr aufwendig, bis das Verständnis für die einzelnen Methoden im ausreichenden Maße vorhanden war. Erschwerend kam hinzu, dass es mehrere Beschreibungen für die gleiche Abschnitte gab, die jedoch für verschiedene Versionen gültig waren. Dadurch wurde es manchmal sehr unübersichtlich, da man immer wieder prüfen musste, ob die Erklärung mit der verwendeten Version übereinstimmt.

Zu der ausbaufähigen Dokumentation kam noch hinzu, dass PACT nicht sehr verbreitet war. Aufgrund der mangelnden Dokumentation mussten immer wieder bestimmte Sachverhalte bzw. Fragestellungen im Internet nachgeforscht werden. Das Ergebnis war oft enttäuschend. Wenn überhaupt gab es nur eine kurze Beschreibung in Form eines Git-Issues. Wobei diese meistens Fehler aus vorherigen Versionen beschrieben, die in der verwendeten Version schon behoben waren. Daher konnten nicht immer Antworten gefunden werden, so dass „trial and error“ erneut als Methode herangezogen werden musste.

Zusammenfassend für die Einbindung von PACT in MARS kann gesagt werden, dass es durchaus simpel und schnell umsetzbar sein kann. Dies setzt allerdings voraus, dass sowohl Kenntnisse darüber bestehen, wie der Consumer und Provider im Detail interagieren und wie die Testfälle in PACT zu implementieren sind.

Im folgenden Bereich werden die einzelnen Untersuchungen bewertet. Zu Beginn kann festgehalten werden, dass die Ausführung der Tests, egal ob es auf der Seite des Consumers oder Providers war, sehr schnell ausgeführt wurden. Dadurch gab es ein schnelles Feedback, ob die Tests erfolgreich waren.

Wie sich aus einzelnen Listings gut ableiten lässt, waren die Meldungen teilweise sehr lang und unübersichtlich. Besonders bei einigen Fehlern wiederholten sich die Fehlerbeschreibungen und sorgten für eine lange Meldungen in der Kon-

sole. Dies erforderte ein genaues lesen, woran der Test gescheitert war. Als gutes Beispiel dient die Meldung aus Listing 1.13 auf Seite 23, hier ist die Beschreibung des Fehlers verwirrend und teilweise falsch. Wie die Fehlerbeschreibung korrekt wiedergibt, fehlte die Map mit den GEO-Daten. Jedoch werden im Diff-Abschnitt auch andere Body-Abschnitte angezeigt, die sich unterscheiden. Dabei ist für diese Abschnitte definiert wurden, dass nur der Typ bzw. Format geprüft werden sollte und keine Überprüfung des Wertes stattfinden sollte. Die komplette Meldung suggeriert aber, dass auch diese Abschnitte fehlerhaft sind. Dies führte zu Verwirrungen und lenkte von dem eigentlichen Fehler ab.

Auch die im Listing 1.12 auf Seite 22 abgebildete Meldung ist unübersichtlich. Zwar gab die Fehlermeldung exakt wieder, dass der Wert von der Eigenschaft „Accept“ im Header nicht stimmte. Jedoch gab es keinen Verweis darauf, dass es sich um den Request-Header handelte. Dazu war nur angegeben, welcher Request nicht empfangen wurde. Eine Ansicht, in der die Unterschiede gegenüber gestellt werden, wäre sinnvoller. Ähnliches gilt auch für das Listing 1.14 auf Seite 25. Die Fehlermeldung war eindeutig beschrieben. Jedoch war auch hier die Ausgabe sehr umfangreich und wiederholte sich zum Teil auch. Besonders die Länge der Meldungen und das teilweise unnötige Anzeigen von Unterschieden machte die Rückmeldungen von PACT sehr unübersichtlich. Dies tritt verstärkt auf, wenn die Tests nicht einzeln ausgeführt wurden, sondern zusammen. Sollten dann mehrere Tests fehlschlagen, kann die Meldung sehr lang werden und dadurch noch unübersichtlicher. Dies kann dazu führen, dass Fehler zunächst übersehen werden oder falsch interpretiert werden.

Als gutes Beispiel für eine gelungene Fehler Ausgabe, kann das Listing 1.15 auf Seite 26 genommen werden. Es ist kurz und knapp beschrieben, wo der Fehler liegt. Dadurch war klar erkennbar, dass der Statuscode in der Response nicht stimmte. Zudem war die Gegenüberstellung der Statuscodes sehr hilfreich.

Ein weiterer negativer Punkt ist die in Listing 1.16 auf Seite 26 abgebildete Meldung. Der Test sollte erfolgreich sein, da nur überprüft werden soll, ob der String im Body der Reponse im Format einer UUID vorliegt. Der zurückgebene String liegt auch im UUID-Format vor. Daher scheint dies ein Fehler innerhalb von PACT zu sein.

Eine Bewertung der Ausgabe bei erfolgreichen Tests muss nicht näher betrachtet werden, da nur die Meldung erscheint, dass die Tests erfolgreich waren. Dies betrifft sowohl die Tests auf Seite des Consumers sowie auf der Seite des Providers.

Abschließend kann über PACT gesagt werden, dass sich die Verwendung durchaus lohnt. Der Vorteil, des schnellen Feedbacks und der durchaus schnellen Implementierung der Tests, überwiegt die Nachteile der teilweise uneindeutigen und/oder zu langen Fehlermeldungen. Auch der zuletzt beschriebene Fehler ändert daran nichts. Mit PACT bekommen Entwickler eine schnelle Rückmeldung, ob ihre Änderungen zu ungewollten Veränderungen der Schnittstellen führen. Weiterhin können sie schnell überblicken, welche Veränderungen bei gewollten Änderungden der Schnittstelle vorzunehmen sind.

Eine Voraussetzung für die Verwendung von PACT sollte jedoch sein, dass die Implementierung der Tests, ähnlich wie bei Unit-Tests, bei den Entwicklern angesiedelt werden sollten. Das bedeutet, die Implementierung sollte von den selben Entwicklern vorgenommen werden, welche auch die Consumer und Provider entwickeln. Es ist sehr hilfreich bei der Implementierung von PACT zu wissen, was genau getestet werden soll und wo die Schwerpunkte liegen. Als Beispiel kann dafür der Header genommen werden. Als Außenstehender ist es schwer zu wissen, ob Eigenschaften überprüft werden sollen und wenn ja, welche genau. Diese Entscheidungen können die Entwickler besser treffen, da sie die Testobjekte detaillierter kennen. Als Alternative könnten Entwickler genau spezifizieren, was getestet werden soll. Mit diesen Angaben kann eine dritte Person die Implementierung von PACT übernehmen. Dabei muss aber die Spezifizierung sehr genau sein, damit die Person, die PACT implementiert, auch genau weiß, was getestet werden muss und wo die Schwerpunkte liegen.

PACT ist durchaus geeignet für MARS bzw. Microservice-Architekturen. Jedoch müssen die oben genannten Voraussetzungen stimmen. Zwar ist die Implementierung in ein bestehendes System umfangreicher als, wenn man PACT von Anfang mit implementiert. Jedoch sind die Vorteile, wie das schnelle Feedback, sehr sinnvoll für ein System, das sich ständig ändert. Denn mit PACT kann sehr gut geprüft werden, ob Änderungen zu ungewollten Veränderungen der Schnittstelle führen oder welche Services noch aktualisiert werden müssen.

4 Offene Aufgaben

In der Masterarbeit sollen die begonnen Arbeiten aus den bisherigen Projekten und Seminaren fortgesetzt werden. Ein Schwerpunkt wird dabei die im Hauptseminar [Pie16] begonnen Ausarbeitung zu dem Testkonzept von Google [WAC12] und das von Toby Clemson erstellte Testkonzept für Programme, die eine Microservice-Architektur verwenden [Cle14]. Ergänzend zu diesen beiden Konzepte sollen weitere aktuelle Testkonzepte gesammelt und verglichen werden. Interessant wird dabei sein, herauszufinden wie andere große Software Hersteller wie z.B. Microsoft zurzeit ihre Software testen. Dies wird unter der Fragestellung stehen: Mit welchen Konzepten wird Software heutzutage getestet?

Diese Konzepte sollen nicht ausschließlich untereinander verglichen werden, sondern auch mit bisherigen Testkonzepten verglichen werden. Dabei soll herausgearbeitet werden, wie sich die Testkonzepte entwickelt haben und ob es einen allgemein gültigen Trend gibt. Weiterhin sollen die Testkonzepte noch unter dem Aspekt untersucht werden, wie gut sie sich für eine Microservice-Architektur eignen könnten.

Ein weiterer Schwerpunkt soll auf der Automatisierung der Testfall-Erstellung liegen. Mit dem Programm Postman ist es schon möglich mithilfe einer Schnittstellen-Beschreibung, wie z.B. einer Swagger-Datei, automatisch Konstrukte für Testfälle zu erstellen. Jedoch müssen die Tests noch mit Daten gefüllt werden. Dazu erstellt Postman nur einen Testfall pro Interaktion. Jedoch wären mehrere Testfälle pro Interaktion sinnvoller, um verschiedene Möglichkeiten und Pfade abzude-

cken. Zudem fehlt den Testfällen ein definiertes Soll-Ergebnis gegen das geprüft werden kann.

Wegen den eben genannten Defiziten ist es interessant zu forschen, inwieweit man diese Defizite beheben kann. Eine mögliche Lösung könnte sein, die Beschreibung der Schnittstelle so zu erweitern, dass fehlende Informationen wie Testdaten und Soll-Ergebnis mit abgebildet werden. In diesem Zusammenhang stellt sich die Frage, ob eine Modell-Beschreibung der Schnittstelle sinnvoller wäre und ob die Beschreibung der Schnittstelle in der Swagger-Datei nicht schon eine Modell-Beschreibung ist. Für den gesamten Aspekt der Automatisierung der Testfall Erstellung können etablierte Konzepte wie das Model-Based Testing sehr nützlich sein. Model-Based Testing beruht auf der Grundlage aus Modellen Testfälle zu erstellen und durchzuführen. Deswegen ist es sinnig, wenn man sich mit der automatischen Testfall Erstellung auseinander setzt, sich in diesem Zuge auch mit Model-Based Testing auseinander zu setzen. Denn Model-Based Testing ist ein etabliertes Konzept, was für viele Probleme, die in diesem Zusammenhang auftauchen können, Lösungen gefunden hat. Diese Lösungen können ggf. wieder verwendet werden oder angepasst verwendet werden.

Literaturverzeichnis

[CK09] Sujit Kumar Chakrabarti and Prashant Kumar. Test-the-rest: An approach to testing restful web-services. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World:*, pages 302–308. IEEE, 2009.

[Cle14] Toby Clemson. Testing strategies in a microservice architecture, November 2014. Zugriff am 30.01.2017 `https://martinfowler.com/articles/microservice-testing/`.

[FB15] Tobias Fertig and Peter Braun. Model-driven testing of restful apis. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1497–1502. ACM, 2015.

[KLC13] Chia Hung Kao, Chun Cheng Lin, and Juei-Nan Chen. Performance testing framework for rest-based web applications. In *2013 13th International Conference on Quality Software*, pages 349–354. IEEE, 2013.

[NCH+14] Alvaro Navas, Pedro Capelastegui, Francisco Huertas, Pablo Alonso-Rodriguez, and Juan C Dueñas. Rest service testing based on inferred xml schemas. *Network Protocols and Algorithms*, 6(2):6–21, 2014.

[pac17] Pact, January 2017. Zugriff am 30.01.2017 `https://docs.pact.io/`.

[Pie16] Alexander Piehl. Testen von microservices. 2016. Ausarbeitung Hauptseminar.

[Pie17] Alexander Piehl. Das testen von microservices. 2017. Ausarbeitung Grundprojekt.

[pos17] Postman, January 2017. Zugriff am 02.01.2017 `https://www.getpostman.com/`.

[PR11] Ivan Porres and Irum Rauf. Modeling behavioral restful web service interfaces in uml. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1598–1605. ACM, 2011.

[Ran12] Karthikeyan Ranganathan. Netflix shares cloud load balancing and failover tool: Eureka!, September 2012. Zugriff am 01.01.2017 `http://techblog.netflix.com/2012/09/eureka.html`.

[Rob06] Ian Robinson. Consumer-driven contracts: A service evolution pattern, 06 2006. Zugriff am 13.12.2016 `http://www.martinfowler.com/articles/consumerDrivenContracts.html`.

[Rod08] Alex Rodriguez. Restful web services: The basics. *IBM developerWorks*, 2008.

[RVG10] Hassan Reza and David Van Gilst. A framework for testing restful web services. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 216–221. IEEE, 2010.

[swa17] Swagger, 01 2017. Zugriff am 02.01.2017 `http://swagger.io/`.

[uHS15] Tobias Bayer und Hendrik Still. Microservices: Consumer-driven contract testing mit pact, 07 2015. Zugriff am 13.12.2016 https://jaxenter.de/microservices-consumer-driven-contract-testing-mit-pact-20574.

[Vin15] Pierre Vincent. Why you should use consumer-driven contracts for microservice integration tests, 03 2015. Zugriff am 13.12.2016 http://techblog.newsweaver.com/why-should-you-use-consumer-driven-contracts-for-microservices-integration-tests/.

[Vit16] Michael Vitz. Consumer-driven contracts – testen von schnittstellen innerhalb einer microservices-architektur, 09 2016. Zugriff am 13.12.2016 https://www.innoq.com/de/articles/2016/09/consumer-driven-contracts/.

[WAC12] James A Whittaker, Jason Arbon, and Jeff Carollo. *How Google tests software*. Addison-Wesley, 2012.