

Hauptprojekt

Alexander Piehl
alexander.piehl@haw-hamburg.de

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

1 Einleitung

2 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. Dabei ist REST zurzeit sehr beliebt bei der Entwicklung von Webservices, da aufgrund von REST Webservices wohl nicht nur leichter zu implementieren sind, sondern auch einfacher zu skalieren sind. [CK09]. Unter Anderem 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 der Ressource muss über eine eindeutige URI gegeben sein. Die Ressourcen sollen zusätzlich über verschiedene Methoden manipuliert werden können. Es müssen mindestens die sogenannten CRUD-Operatoren zur Verfügung stehen. CRUD steht für Create, Read, Update und Delete und beschreibt die grundsätzlichen Daten Operationen. Bei Rest werden dafür die standardisierten HTTP-Methoden verwendet, welche im Standard RFC 2616 definiert wurden sind [KLC13]. In der Tabelle 1 auf Seite 1 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, wie z.B. 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 den Zustand notwendig sein, müssen die Clients die Informationen mitgeben. Dahingehend ist es auch mit REST möglich kompliziertere Programmmzustände abzubilden [PR11].

Die jeweiligen Nachrichten können in verschiedenen Formate vorliegen [RVG10]. Sehr häufig werden XML oder JSON oder beide Formate für die Nachrichten verwendet. Eine Vorschrift existiert 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 der Ressourcen ist getrennt von den Ressourcen. Dies bedeutet Ressourcen können in verschiedenen Formate vorliegen, wie JSON und XML.
- **Uniform Interface:** Auf die Ressourcen wird nur über standardisierten HTTP-Methoden zugegriffen.
- **Statelessness:** Jede Kommunikation erfolgt Zustandslos.

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

Was ist besonders an Rest? Unterschiede zu SOAP? Warum lieber Rest als SOAP? Leichtgewichtig: leichtes ändern der Schnittstelle möglich

2.1 Besonderheiten beim Testen ?

Die Kommunikation zwischen Consumer und Service erfolgt nach einem festgelegten Kontrakt, welcher auch häufig Schnittstelle genannt wird. In dem Kontrakt ist festgelegt, wie die Kommunikation zwischen Consumer und Service aussieht. Das heißt, mit dem Kontrakt ist definiert welche Methoden der Service seinen Consumer anbietet und wie die Consumer sie aufrufen können, also welche Parameter benötigt werden und in welchem Format der Consumer seine Antwort erhält. Dahingehend ist es enorm wichtig, dass der Kontrakt eingehalten wird. Denn ansonsten kann es zu Fehlern führen.

Des Wegen muss ein Schwerpunkt beim Testen von Anwendungen, welche REST verwenden, darauf liegen, zu kontrollieren, dass der Kontrakt weiterhin gültig ist. Daher muss geprüft werden, dass Veränderung am Consumer bzw. Service nicht zur einer Verletzung des Kontraktes führen.

Besonders bei Anwendungen mit einer Microservice-Architektur, bei der die Kommunikation hauptsächlich über das Netzwerk geschieht, ist es von großer

Bedeutung, dass die Kommunikation funktioniert und nicht auf Grund fehlerhaften Kontrakten zu Fehlern kommt.

Wegen der Verlagerung der Kommunikation in das Netzwerk, wird das Überprüfen der Kontrakte umfangreicher, da schlicht und ergreifend sehr viele Kontrakte vorliegen. Ergänzend dazu sind die einzelnen Services gleichzeitig Consumer und Service Provider. Daher kann es schnell unübersichtlich werden, wer welchen Services anfragt und wie.

Besonders bei gewünschten Änderungen eines Kontraktes kann es unübersichtlich werden. Aufgrund der möglichen Vielzahl von Consumer, kann sehr schnell der Überblick verloren werden, welche Consumer aktualisiert werden müssen und welche nicht. Dies kann zur Folge haben, dass ein Service Methoden oder Variante einer Methode bereitstellt, die nicht mehr benötigt werden.

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

3 Consumer Driven Contract Test

Bei dem Ansatz Consumer-Driven Contracts werden die Schnittstellen bzw. die Verträge aus der Sicht des Clients/Consumers definiert [Rob06]. Dabei spielt es keine Rolle, ob es nur einen Consumer oder mehrere existieren. Die jeweiligen Anfragen und gewünschten Antworten werden notiert und auf Grundlage dieser Informationen wird der Service implementiert. Dabei baut der Ansatz auf der Grundannahme auf, dass der Consumer am besten wüsste, was er nutzen möchte.

Wie sich aus der generellen Beschreibung dieses Ansatzes ableiten lässt, eignet sich dieser Ansatz am Besten, wenn sowohl die Consumer wie der Service neu implementiert werden. Bei bestehenden Anwendungen könnte dieses Verfahren unter anderem dafür genutzt werden, um zu überprüfen, ob der Service 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 Services haben. Andernfalls könnte der Service nur erraten, wie er die jeweiligen Anforderungen umsetzen muss. Durch die eindeutige Definierung der Anforderungen, kann der Service so schlank wie möglich implementiert werden.

Auf der Basis dieses Konzeptes setzen die Consumer Driven Contract Tests auf. Ein Tool, welches Consumer Driven Contract Tests unterstützt ist PACT. PACT ist ein Open-Source Tool, welches auf Github verwaltet wird.

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

Mit dem Tool PACT können die Consumer Driven Contract Tests in verschiedenen Programmiersprachen geschrieben werden. Zur Auswahl stehen dabei Ruby, C#, JavaScript und Java. Im nächsten Kapitel wird der generelle Ablauf von PACT anhand eines Beispiels erläutert, bevor im darauffolgenden Kapitel die Vorteile beschrieben, 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.

3.1 Ablauf Consumer Driven Contract Test

In diesem Kapitel wird der generelle Ablauf von Consumer Driven Contract Tests beschrieben. Der Ablauf wird anhand mehrere Beispiele erläutert, welche von verschiedenen Seiten stammen, die Consumer Driven Contract Tests erläutern [uHS15, Vit16, Vin15].

Wie bereits erläutert basiert Consumer Driven Contract Test auf Consumer Driven Contract. Daher wird auch bei Consumer Driven Contract Test zunächst der Consumer bzw. die Consumer implementiert. In diesem, Beispiel fragt der Consumer Produktdetails über ein bestimmtes Produkt beim Provider an.

Nach dem die Implementierung des Consumers soweit erfolgt ist, dass er Anfragen stellen und verarbeiten kann, kann PACT für den Consumer implementiert werden. Der größte Aufwand bei PACT ist die Implementierung der Testfälle auf der Seite des Consumers.

Für die Implementierung von PACT auf der Seite vom Consumer 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. Der Hintergrund dafür ist, dass man damit mehrere PACT Tests für unterschiedliche Provider und Consumer definieren kann.

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

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

Über die Methode `createFragment` werden wie Aufrufe des Consumers mit den zu erwartenden Antworten erstellt. Dies geschieht mit einer von PACT definierten DSL.

```
@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 details\"}")
        .toFragment();
}
```

Zunächst wird eine Map erstellt, in der verschiedene Optionen für den Header definiert werden können. 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. Diese `PactFragment` wird vom `PactDslWithProvider` erstellt, welches der Methode als Parameter übergeben wird. Da damit das Fragment erstellt, bekommt es den passenden Namen `builder`. Mit dem Builder können die verschiedenen Interaktionen definiert werden.

- **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.
- **status:** Soll der Statuscode überprüft werden, kann mit dieser Methode der zu erwartende Statuscode definiert werden.
- **body:** Mit der `body` 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 mithilfe des Builders erstellt werden. Jeder Interaktion beginnt mit der `uponReceiving`. Um eine weitere Interaktion hinzuzufügen, kann man einfach nach der Definition der zu erwartenden Antwort einfach mit der Methode `uponReceiving` eine neue Interaktion hinzugefügt

werden, welche nach dem gleichem Schema abläuft. In diesem Beispiel wäre dies nach der Methode body.

Sobald das PactFragment, welches die Interaktionen enthält, implementiert ist, kann getestet werden, ob der Consumer selber den Kontrakt einhält. Dafür wird die Methode runTest umgesetzt.

```
@Override
```

```
protected void runTest(String url) {  
    URI productDetailsUri = URI.create(String.format("%s/%s/%s", url, "productd  
  
    ProductDetailsFetcher productDetailsFetcher = new ProductDetailsFetcher();  
    ProductDetails productDetails = productDetailsFetcher.fetchDetails(productI  
    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. Einerseits wird getestet, ob der Consumer eine korrekte Anfrage erstellt und andererseits, ob der Consumer mit der Antwort vom Provider Stub zurecht kommt. Dies wird unter anderem mithilfe von asserts gelöst.

Sobald alle Methoden korrekt implementiert wurden sind, kann die Klasse ausgeführt werden. Entweder kann die Klasse direkt gestartet werden oder man führt die Tests mithilfe eines Maven/Gradle-Plugins von PACT aus.

Sind die Tests erfolgreich durchgelaufen, wird automatisch das sogenannte PACT-File erstellt.

```
{  
  "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 befinden sich die jeweiligen Provider und Consumer Namen drin sowie die definierten Interaktionen und die zu erwarteten Antworten. Ergänzend dazu ist im PACT-File noch angegeben mit welchen Versionen, die Datei erstellt worden ist.

Mit dem eben nun erstellten PACT-File kann der Provider getestet werden. Dafür muss der Provider entweder das Gradle oder Maven Plugin von PACT für den Provider verwenden. Um das Plugin verwenden zu können, muss einmal der Pfad zum PACT-File angegeben werden. Das PACT-File kann lokal vorliegen, es kann aber auch via eine URL aufgerufen werden. Zusätzlich muss die Adresse des Providers angegeben werden. Die entsprechenden Einstellungen am Beispiel der Verwendung von Gradle sieht so aus:

```

pact {
  serviceProviders {
    productDetailsServiceProvider {
      protocol = 'http'
      host = 'localhost'
      port = 10100
      path = '/'
      hasPactWith( 'productServiceConsumer' ) {
        pactFile = file( "../product-service/target/pacts/Product_Service-Pro
      }
    }
  }
}

```

Damit der Test ausgeführt werden kann, muss der Provider gestartet werden. Sobald dies geschehen ist, kann das Plugin für den Provider ausgeführt werden. Das Ergebnis des Tests werden in dann in der Konsole angezeigt.

- Implementierung Consumer
- Definieren der Aufrufe
- PACT für Consumer vorbereiten
 - createFragment

- providerName
- consumerName
- runTest
- erstelltes PACT-File
- Verfiy mit Maven Plugin des Providers
- weiteres Vorgehen

3.2 Vorteile Consumer Driven Contract Test

Neben den generellen Vorteilen, welche man sich von dem Ansatz Consumer Driven Contract Test verspricht und schon im vorherigen Kapitel kurz skizziert wurden sind, gibt es noch weitere Vorteile, die sich explizit auf das Testen beziehen.

In der vorherigen Ausarbeitung zum Grundprojekt wurden die Anforderungen beschrieben, welche beim Testen einer Microservice Anwendung vorliegen. Eine dieser Anforderung war, dass das es vom Testen ein schnelles Feedback gibt. Genau diese Anforderung soll Consumer Driven Contract Test erfüllen. Jede noch so große Änderung am Service soll ohne großen Aufwand gegen das PACT-File getestet werden können. Dadurch soll man ein schnelles Feedback bekommen, ob der Service den Vertrag noch einhält.

Ergänzend dazu verspricht Consumer Driven Contract Test einen feinkörnigen Einblick darin, was die Änderung am Service für Auswirkung hat.

Dazu soll man schnell ein Überblick bekommen, ob die Änderung am Service zu unerwarteten Fehlern führt. Aufgrund des zügigen Feedbacks könnte das Fehlverhalten schnell korrigiert werden.

Bei gewünschten oder notwendigen Änderungen am Service, weil ein neuer Consumer hinzugefügt wurden ist oder es sich bei einem Consumer die Anforderungen an den Service geändert haben. soll man aufgrund von Consumer Driven Contract Test schnell erkennen, ob aufgrund der Änderung andere Consumer angepasst werden müssen.

Zusammenfassend betrachtet lassen sich die vermeintliche Vorteile von Consumer Driven Contract Test dahingehend vereinen, dass es ein schnelles Feedback gibt, welche Auswirkung die Änderung am Service auf den Kontrakt hat. Dadurch können ungewollte Fehler schnell identifiziert werden oder notwendige Aktualisierungen von Consumer festgestellt werden.

Für ein System wie MARS mit einer Microservice-Architektur bei der das Messaging im 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 zu deployen. Consumer Driven Contract Test ist dabei sehr vielversprechend diesen Prozess zu begleiten, da es schnelles Feedback geben soll, welche Auswirkungen die Änderungen haben. Ob Consumer Driven Contract Test die beschriebenen Vorteile einhält, wird im nächsten Abschnitt behandelt, bei dem es um die Umsetzung von Consumer Driven Contract Test innerhalb von MARS geht.

- Vorteile bei Änderungen im Service kann sofort wieder gegen die Datei getestet werden, um zu prüfen, ob alle Consumer noch korrekt funktionieren
- übersicht, was die Consumer Anfragen
- schnelles Feedback
- wissen, ob Änderung zu ungewollten Fehlern
- wissen, welche consumer aktualisiert werden müssen
- MARS System, welches regelmäßig angepasst wird
- Architektur, welche Fokus auf das Netzwerk hat
- Fehler wegen Schnittstelle problematisch

Was verspricht man sich davon? Verbindung zu MARS - System im Wandel

- Erklärung Consumer Driven Ansatz
 - Vorteile dazu gibt es einen feingründigen Einblick und schnelles Feedback für das Planen von Änderungen. Ergänzend dazu können gezielt einzelnen Consumer angesprochen werden.
- Erläuterung Consumer Driven Contract Test
 - Tool PACT
 - Consumer definiert seine Anfrage und die zu erwarteten Antworten, samt UNIT-Test
 - Implementierung im Consumer
 - Beim Ausführen dieser Tests wird ein Server gestartet, der mit den entsprechenden Antworten auf die Anfragen reagiert
 - Dabei wird ein File erstellt, welches die Aufrufe und Antworten enthält
 - mit diesem File wird nun der Service Provider getestet
 - Antwortet er auf die Anfragen korrekt, wie es im File beschrieben ist

4 Consumer Driven Contract in MARS

Nach der grundlegenden Befassung mit PACT soll nun geprüft werden, ob sich PACT für MARS anbietet. Wie schon im Kapitel zu den vermeintlichen Vorteilen von PACT beschrieben, könnte PACT einige Anforderungen an die Tests zu erfüllen. Kurz zusammen gefasst geht es um ein schnelles Feedback der Tests, besonders bei Änderungen am Source Code sowie eine dementsprechendes aussagekräftiges Feedback.

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

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, ob die Verwendung von PACT innerhalb von MARS sinnvoll ist.

Für die Erstellung der Machbarkeitsstudie wurde der Bereich der Imports in MARS ausgewählt. 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 übersichtlich. Dazu ist man in der Lage diesen Bereich des Systems schnell mit Testen zu befüllen. Für die exakte Implementierung von PACT werden die Interaktionen zwischen dem Metadata-Client (Consumer) und dem Provider Metadata verwendet.

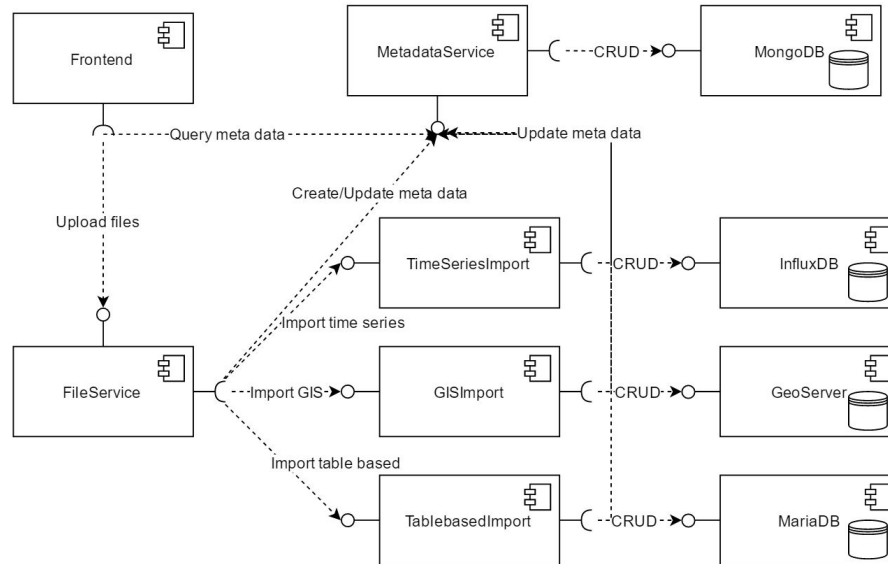


Abb. 1. Diagramm der Komponenten für die Import Services

Wie in der Abbildung 1 zu den Import Services gut erkenntlich interagieren alle Services, die für den Import zuständig sind, mit dem Service Metadata. Der Service Metadata hat die Aufgaben für die einzelne Dateien, die importiert werden, Metadata 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 dem Service Metadata im Objekt Metadata-Client gekapselt. Wenn eine Interaktion mit dem Service Metadata erforderlich ist, wird das Objekt Metadata-Client erzeugt, welches dann die Interaktionen übernimmt.

4.1 Testumgebung

Die verwendete Testumgebung ist sehr ähnlich zu der Testumgebung 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 wurde.

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 und zwar wurde von den Import-Projekten ein Branch erstellt. Zum Zeitpunkt der Untersuchung von PACT innerhalb von MARS, gab es laufend 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 keine Änderungen diesen Prozess beeinträchtigen können.

Für die Implementierung von PACT wurde IntelliJ als Entwicklungsumgebung verwendet. Dazu wurde Maven als Build-Management-Tool verwendet, da alle MARS-Projekte Maven verwenden und eine Umstellung auf Gradle zu viel Ressourcen verbraucht hätten, ohne vornhinein ermitteln zu können, ob es sich dadurch Vorteile ergeben. Ergänzend dazu wurden die Entwicklungswerkzeuge von Chromium und das Programm Postman verwendet, um die Interaktionen besser zu verstehen und MARS mit default Werten zu füllen. Zusätzlich wurde ein Framework für die Erstellung von Mocks verwendet und zwar das Framework Mockito.

4.2 Einbindung in MARS

Zwischen dem Metadata-Client und Metadata gibt es mehrere verschiedene Interaktionen. Dabei sind alle CRDU-Methoden, also GET, POST, PUT und DELETE, vertreten. Bei der Implementierung von PACT wurden jedoch nicht alle Interaktionen umgesetzt, da es zu aufwendig gewesen wäre. In der Tabelle 2 auf der Seite 11 sind die Interaktionen kurz beschrieben, welche mit PACT implementiert werden sollten.

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 wurden der Zustand genommen, der bei der Erstellung des Branches vorlag. Mithilfe dieses Soll-Zustandes wurden die Interaktionen in PACT implementiert. Als vorbereiteter Schritt davor, mussten die

Interaktionen genauer analysiert werden, damit klar definiert werden kann, wie die Anfrage und die dazu gehörige Antwort auszusehen hat.

Bei MARS werden die APIs mithilfe von Swagger[swa17] dokumentiert. Swagger ist ein open source Framework, welches beim Entwerfen, bei der Erstellung und Dokumentation unterstützen soll. Swagger erstellt für jede API eine extra Datei, in welcher die Schnittstellen beschrieben werden. Die Datei heißt meistens `swagger.yaml`. An der Endung der Datei kann man erkennen, dass es sich die Datei im YAML-Format vorliegt. In dieser Datei sind alle Methoden beschrieben, die vom Consumer angefragt werden können. Zu den einzelnen Methoden sind wichtige Informationen angegeben, wie z.B. der Pfad, die Parameter, die erwartet werden. Zu den Parameter ist noch angegeben, welcher Typ vorliegen soll und wie sie mit geschickt werden sollen, also ob sie z.B. im Body als Query im Pfad mitgeschickt werden sollen. Die Swagger-Datei ist sehr hilfreich für das Programm Postman.

Postman ist ein Programm, welches ähnlich wie Swagger bei der Entwicklung und Dokumentation von Schnittstellen unterstützen sollen. Bei Postman liegt der Fokus aber auf dem Testen von Schnittstellen und außerdem fügt es im Gegensatz zu Swagger über eine grafische Oberfläche. Mit Postman können Anfragen an einen Provider gestellt werden, um im Anschluss die Antwort zu überprüfen.

Die mit Swagger erstellte Datei kann in Postman importiert werden und Postman erstellt auf Grundlage der Informationen aus der Swagger-Datei automatisch Testfälle. Jedoch fehlt diesen Testfällen Bedingungen, ob sie erfolgreich sind. Diese Bedingungen können mit einer von Postman definierten DSL selbst implementiert werden. Postman bietet dafür Templates, um standardmäßige Überprüfungen auf Status Code oder ähnliches schnell umsetzen zu können.

Da leider manche Informationen in der Swagger-Datei fehlten oder veraltet waren, mussten die automatische Tests manuell aktualisiert werden. Dafür waren die Entwicklungswerkzeuge vom Browser Chromium bzw. Chrome sehr hilfreich. Denn die einzelnen Interaktionen können einem Tool aufgezeichnet und analysiert werden. Dafür wurden die jeweiligen Interaktionen über das Frontend aufgezeichnet und anschließend ausgewertet. Mithilfe dieser Informationen wurden die Testfälle innerhalb von Postman aktualisiert. Sobald alle Tests in Postman implementiert wurden sind und erfolgreich abliefen, wurde damit begonnen PACT auf der Seite des Consumers einzubinden. Jedoch gab es ein Problem mit der DELETE-Methode. Beim Testen mit Postman konnte der Aufruf nicht ausgeführt werden. Der Service antwortete mit dem Statuscode 405. Dies bedeutet, dass die gewählte Methode nicht erlaubt ist. Die Methode wurde jedoch in PACT implementiert, da es sich um einen Bug handelt. Dahingehend wurde die Implementierung der DELETE-Methode in PACT alleine auf Grundlage der Swagger-Datei ausgeführt.

Implementierung des Consumers Bei der Implementierung von PACT beim Consumer wurde so ähnlich vorgegangen, wie im allgemeinen Beispiel beschrieben. Auch hier wurde zunächst eine neue Klasse erstellt und zwar 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 Interaktionen 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 in 1.1 abgebildet.

Listing 1.1. Test

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

Wie in der Erklärung von REST erwähnt, muss jede Ressource über eine eindeutige URI erreicht werden. Bei den Metadata Einträgen erfolgt dies über die Eigenschaft DataID. Daher wird dem Pfad einfacher der Wert DataID angehängt. Weitere Informationen sind in der Query und Body nicht notwendig. Im Header ist nur definiert wurden, dass nur JSON akzeptiert wird. Der Provider soll auf diesen Aufruf mit dem Statuscode 200 antworten. Obwohl ein Header für die Antwort definiert ist, findet keine Überprüfung statt, da der Header keine Werte enthält.

Bis auf die einzelnen Werte unterscheidet sich dieses Fragment nicht vom generellen Beispiel. Nur der Body, welcher die Antwort enthalten soll, ist deutlich umfangreicher. Dafür wird beim Body die Methode createJsonBody aufgerufen, die die zu erwartende Antwort im Body definiert. Diese Methode ist abgebildet in 1.2. Ähnlich wie bei der Erstellung des Fragments wird auch für die Erstellung des Body eine PACT definierte DSL genutzt.

Listing 1.2. Test

```
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 1.2 gut ableiten kann, ist das Metadata-Objekt recht umfangreich. Da bei der Anfrage definiert wurde, dass nur JSON akzeptiert wird, muss die Antwort selbstverständlich 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. Die einzelnen Methoden fügen nicht nur die Eigenschaften hinzu, sondern definieren auch die jeweiligen Matcher mit. Generell wird der Typ der jeweiligen Eigenschaften bestimmt und sollte kein Wert angegeben werden, wird ein zufälliger Wert vom festgelegten Typ erstellt. Dazu muss immer der Name der jeweiligen Eigenschaft mitangegeben werden. Die bei der Erstellung des JSON-Bodys verwendeten Methoden sind in der Tabelle 3 auf Seite 14 näher beschrieben.

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

Tabelle 3. Erklärung der verwendeten PACT DSL Methoden

Der damit definierte Body soll die in abgebildete Antwort darstellen. Werte werden nur überprüft, wenn sie aufgrund der Anfrage sicher vorliegen sollte.

Wenn die Metadata von einer entsprechenden DataID abgefragt werden, dann muss sich diese DataID auch in der Antwort befinden. Da zufällige Werte erzeugt werden, wenn kein Wert definiert ist, muss entsprechend der Null-Wert übergeben werden, wenn die Eigenschaften ohne Wert erwartet wird.

Nun muss für diese Interaktion noch die Methode `runTest` implementiert werden. Der dazu passende Code-Abschnitt ist in 1.3 auf Seite 15 abgebildet.

Listing 1.3. Test

```
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. Eureka [Ran12] ist ein von Netflix entwickeltes Programm für die Service-Discovery. Vereinfachtesagt 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. Denn für die Erstellung eines Objektes vom Typ `MetadataClient` wird der `EurekaClient` 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 gestubben Provider. Bisher war es nicht möglich diese Information in der gemockten `EurekaClient` unterzubringen. Daher wurden der `MetadataClient` um Methoden ohne 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. Dies Umstellung betrifft alle Interaktionen.

Nachdem URI zusammengebaut wurden ist, kann diese URI dem Client übergeben werden. Die damit aufgerufene Methode wird die Metadata zurückliefern, wie im vorher im Fragment definiert worden ist. Deswegen muss diese zurückbekommene Metadata die selbe DataID enthalten, wie bei der Anfrage mitgeschickt wurden ist. 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 eben gerade beschriebenen Interaktion. Nur der Body der zu erwartenden Antwort unterscheidet sich. Es wird nun Array von Metadata-Objekten erwartet. Die für die Erstellung dieses Array zuständige Methode ist verkürzt in 1.4 auf Seite 15 dargestellt.

Listing 1.4. Test

```
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. Das hier definierte Objekt ist das gleiche wie in 1.2 auf Seite 13 dargestellt.

Auch in der `runTest`-Methode gibt es keine großen Unterschiede. Es wird selbstverständlich eine andere Methode vom Metadata-Client aufgerufen, die eine Liste von Metadata-Objekten zurückgibt. Ergänzend dazu ist die `assert`-Methode passend angepasst.

Als nächstes wird die PUT Interaktion implementiert. Diese unterscheidet sich durchaus von den bisherigen Implementierungen. Neben den Unterscheidungen im Header der Anfrage, muss für diese Interaktion die 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, der in der Antwort stehen soll, 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 auch 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 durchaus mehr 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 200. Der Statuscode 201 gibt wieder, dass ein neuer Eintrag erfolgreich erstellt wurde. Im Body der Antwort wird eine UUID als Plain-Text erwartet. Diese UUID wird als `DataID` verwendet.

Der Aufruf in der `runTest`-Methode unterscheidet sich dahingehend, dass der Methode im Metadata-Client die einzelnen Informationen zum MetadataEintrag mit übergeben werden muss. Da der genaue Wert der UUID nicht vorher gesagt werden kann, wird kein Assert auf die UUID übernommen. Es wird nur geprüft, dass eine UUID zurück geliefert. Diese Überprüfung übernimmt PACT.

Die DELETE-Interaktion ist sehr ähnlich wie die erste Interaktion, die implementiert wurden ist. Anstatt der GET-Methode wird die DELETE-Methode verwendet und es wird kein Body in der Antwort erwartet.

Der Aufruf in `runTest`-Methode unterscheidet sich einmal 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 der Seite des Consumers ausgeführt werden

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 deutlich übersichtlicher als auf der Seite des Consumers. Beim Provider muss, wie oben in der allgemeinen Beschreibung schon vorgestellt, nur das Plugin eingebunden. Da MARS Maven verwendet, wird das Maven-Plugin eingebunden. In der 1.5 auf Seite 17 ist der Abschnitt der pom.xml vom Metadata-Service abgebildet, die für das PACT-Plugin benötigt wird.

Listing 1.5. Test

```
<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>
```

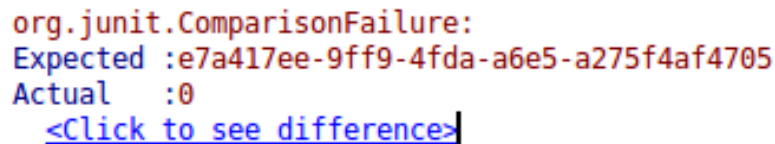
In dem Auschnitt aus der pom.xml findet sich unter anderem die Namen vom Provider und Consumer wieder. Zudem muss die Adresse angegeben werden unter denen der gestartete Service gefunden werden kann. Wie schon in der allgemeinen Beschreibung von PACT erwähnt, kann PACT den Provider nur testen, wenn er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

4.3 Untersuchung

Mit der fertigen Einbindung von PACT konnten mehrere 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 nur die Testfälle beschrieben und die jeweiligen Ergebnisse skizziert. Die Einordnung und Bewertung der Ergebnisse finden jedoch erst im nächsten Kapitel statt.

Beim ersten Testfall gibt der Consumer das Ergebnis vom PACT erstellten Provider nicht korrekt wieder. Über die GET-Anfrage mit der 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 sollte geprüft werden, wie PACT mit der Rückgabe einer falschen DataID um geht.

Da in der runTest-Methode ein Assert auf die DataID implementiert wurden ist, schlägt dieses fehl. Die dazugehörige Meldung ist in der Abbildung 2 auf Seite 18 abgebildet. Dieser Fehler wurde jedoch nur aufgrund des vorher definierten Asserts gefunden. Sollte weitere Fehler im Metadata-Objekt vorhanden sein, werden diese nicht gefunden.



```
org.junit.ComparisonFailure:
Expected :e7a417ee-9ff9-4fda-a6e5-a275f4af4705
Actual   :0
<Click to see difference>|
```

Abb. 2. Fehlgeschlagenes Assert aufgrund der falschen DataID

Bei get a Metadata Entry den Rückgabewert von getDataId manuell auf String 0 angepasst, Assert schlägt fehl. Weitere Daten können nur überprüft werden, wenn man die Testdaten kennt. Header und ähnliches übernimmt Spring mit dem RestTemplate

Bei einem weiteren 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 in 1.6 auf 18 dargestellte Meldung, wobei der Stacktrace gekürzt wurde.

Listing 1.6. Fehler im Header

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

..

```
au.com.dius.pact.consumer.PactMismatchException: Pact verification failed for th
createMetadata:
    HeaderMismatch - Expected header 'Content-Type' to have value 'application/j
    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)
```

Für das Testen des Provider wurden mehrere verschiedene Testfälle erstellt. Beim ersten Test sollte der Provider 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, dass dies wahrscheinlich ein seh häufiger Fehler ist, der beim Schnittstellen Test auftaucht. Selbst kleine Änderung an der Schnittstelle oder an dem Objekt, welches angefragt wird, kann dazu führen, dass das angefragte Objekt bei der Antwort anders aussieht. Durch den Ausfall des Geoservers enthielten alle Metadata-Einträge keine Geo-Informationen, welcher in der MAP `additionalTypeSpecificData` gespeichert werden. Diese Map fehlte also alle Metadata-Einträgen. Dieses Fehlen produzierte die auf Seite 19 dargestellte 1.7 Fehlermeldung.

Listing 1.7. 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": "CdWPkGzoyOIPJJbjZah",
+   "description": "GIS_Import_via_Postman",
  "errorMessage": null,

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

```

        "records": null,

@19
        "state": "UPLOADED",
-       "title": "NPQsfuCfleUsNkKmncczm",
+       "tags": null,
+       "title": "Postman_Import",
        "type": "GIS",

@21
        "type": "GIS",
-       "userId": 994762534
+       "userId": 1
    }

```

Im nächsten Testfall für den Provider wurde wieder der Header manipuliert, aber dieses mal jedoch auf der Seite des Response. Für diesen Testfall wurde die PUT-Interaktion ausgewählt, die den Wert aktualisiert, welcher den aktuellen Zustand des Uploads beschreibt. Dafür wurde der Fehler im PACT-Fragment gesetzt und nicht der Metadata-Service angepasst, da dies deutlich aufwendiger gewesen wäre. Für den Response-Header wurde die Eigenschaft auf "Content-Length" auf "1" gesetzt, da der Response Body auf die PUT-Abfrage leer ist, ist die Content-Length auch null. Dies führte zu der auf Seite 21 in Listing 1.8 aufgeführten Meldung.

Listing 1.8. Test

```

Verifying a pact between metadata-client and metadata-service
[Using file /home/alex/Mars/mars-import/metadata-client/target/pacts/metadata-
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 sta
[Using file /home/alex/Mars/mars-import/metadata-client/target/pacts/metadata-
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** status
Expected header 'Content-Length' to have value '1' but was '0'

Beim nächsten Testfall sollte der Statuscode nicht mit dem vorher definierten Statuscode übereinstimmen. Dieser Fehler kann schnell auf der Seite des Providers auftauchen, wenn der Provider mit der Anfrage nicht umgehen kann oder einfach 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 sollte, welcher gar nicht vorhanden ist. Daher gibt der Provider den Statuscode 404 zurück und nicht wie im PACT-Fragment definiert 200. Dadurch kommt die in Listing 1.9 auf Seite 22 abgebildete Meldung.

Listing 1.9. Test

```
Verifying a pact between metadata-client and metadata-service
[Using file /home/alex/Mars/mars-import/metadata-client/target/pacts/metadata-
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
assert expectedStatus == actualStatus
 | | |
 200 | 404
 false

Der letzte Testfall war in dem Sinne gar nicht vorgesehen wie die anderen beschriebenen Testfälle. Beim Erstellen eines neuen Metadata-Eintrags mithilfe der POST-Interaktion, soll eine DataID, welche im UUID-Format vorliegen soll, als einfacher Text zurück gegeben werden. Dafür wurde auch 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 für den Response Body so definiert worden, dass keine exakte Überprüfung des Wertes stattfindet, sondern eine Überprüfung über reguläre Ausdrücke. Mit der Überprüfung via reguläre Ausdrücke wird nur getestet, ob der zurück gegebene Wert im UUID-Format vorliegt. Obwohl das Fragment so für diese Überprüfung so erstellt wurde, wie es in der PACT Dokumentation beschrieben wurden ist, erschien die auf Seite 23 in Listing 1.10 abgebildete Fehlermeldung. Wie man der Meldung entnehmen kann, schlug die Überprüfung fehl, da die DataIDs unterschiedlich ist. In der Dokumentation gab es keine weitergehende Informationen zu diesem Verhalten.

Listing 1.10. Test

```
Verifying a pact between metadata-client and metadata-service
[Using file /home/alex/Mars/mars-import/metadata-client/target/pacts/metadata-
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
/ -> mismatch

Diff:

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

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

- Beschreibung der Testumgebung
- Ziel beschrieben
- Auffinden eines Services und Consumers
- Aufrufen herausfinden via Browser und Postman + Swagger
- Testen der Aufrufe via Postman, um festzustellen, dass sie funktionieren und korrekt sind
- Aktueller Stand als korrekt anzusehen und Soll-Zustand
- Branch
- Implementierung des Consumers

4.4 Bewertung der Ergebnisse

Das Fazit und die damit verbundene Bewertung für PACT soll in zwei grobe Bereiche unterteilt werden. Einmal 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 dazu kam, dass man selber weder den Metadata-Client noch den Metadata-Service implementiert hatte. Im generellen Ablauf von PACT wird jedoch davon ausgegangen, dass man Consumer sowie den Provider selber implementiert. Dieser Umstand führte zu einigen Schwierigkeiten. Denn es waren nicht immer alle notwendigen

Informationen vorhanden. Zwar lieferten die Swagger-Dateien sehr wichtige Informationen, jedoch waren diese nicht immer korrekt oder waren veraltet. Zudem fehlten auch einige Informationen. Auch wenn man diese Informationen teilweise über die Analyse der Aufrufe via Chrome herausarbeiten konnte, war dies jedoch mit deutlichem Mehraufwand verbunden. Dieser Mehraufwand wäre vermutlich nicht vorhanden, wenn man das System genau kennen würde.

Aufgrund der fehlenden Informationen konnte man auch nicht immer entscheiden, ob das aktuelle Verhalten korrekt ist. Deswegen gab es teilweise auch keine andere Möglichkeit als das aktuelle Verhalten als das Soll-Verhalten zu definieren, um sich auch nicht selber aufzuhalten.

Um die einzelnen Interaktionen besser zu verstehen, war es unumgänglich die Interaktion zuerst in Postman umzusetzen. Denn damit bekam 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 war die Implementierung von PACT auch nicht ganz ohne Schwierigkeiten. Schaut man sich zuerst beschriebene generelle Beispiel und die danach erfolgte Implementierung in MARS an, sieht es auf den ersten Blick sehr kompakt und vor allem simpel aus. Im Prinzip ist es auch simpel und kompakt, wenn man

- Erklärung
- Erläuterung zu der Verbindung mit Microservice
- Normale Implementierung
- Einbinden in Mars
- Fazit

5 Ausblick Masterarbeit

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.
- [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.
- [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 (IT-NG), 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. Zu-

griff am 13.12.2016 <https://www.innoq.com/de/articles/2016/09/consumer-driven-contracts/>.