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 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 Programmzustände 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.
- **Statelessnes:** 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 Cosumer 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 Grundnahme 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 erahnen, wie er die jeweiligen Anfoderungen 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 erwartetenden Antworten erstellt.

```
@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\":\"Thisis the description for pr
            .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 des Builder können die verschiedenen Interaktionen definiert werden.

- **uponReciving:** Mit dieser Metohde 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 erwartetende 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 erwartetende 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 uponReciving. Um eine weitere Interaktion hinzufügen, kann man einfach nach der Definition der zu erwartetenden Antwort einfach mit der Methode uponReciving 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 anderseits , ob der Consumer mit der Antwort vom Provider Stub zurecht kommt. Dies wird unter anderm 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 erwarteteden 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 gestestet 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 Microserive 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 eine 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 Diven 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
- sachnelles Feedback
- wissen, ob Änderung zu ungewollten Fehlern
- wissen, welche consumer aktualisiert werden müssen
- MARS System, welches regelmäßig angepoasst wird
- Architektur, welche Fokus auf des 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 feinjörnigen 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, samit UNIT-Test
  - Implmentierung 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

4.1 Testumgebung

4.2 Einbindung in MARS

4.3 Ergebnisse

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

4.4 Fazit

- 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.

[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.

[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/`.