Fakultät für Informatik und Mathematik 07

Bacheloararbeit

über das Thema

Generative Testerstellung für Microservice-Architekturenit

Autor: Fabian Wilms
holtkoet@hm.edu

Prüfer: Prof. Dr. Ulrike Hammerschall

Abgabedatum: xx.xx.2017

I Kurzfassung

kurzfassung

Abstract

Das ganze auf Englisch.

II Inhaltsverzeichnis

III Abbildungsverzeichnis

IV Tabellenverzeichnis

V Listing-Verzeichnis

VI Abkürzungsverzeichnis

LHM	Landeshauptstadt München
LDAP	Lightweight Directory Access Protocol
HATEOAS	Hypermedia as the engine of application state
RPC	Remote Procedure Call
REST	Representational State Transfer

1 Einführung und Motivation

IT nimmt sowohl im privaten als auch geschäftlichen Alltag eine immer größere Rolle ein. Die Übernahme von Bereichen, die ehemals als nicht durch Computer austauschbar erachtet wurden, schreitet immer weiter fort. Doch dadurch steigen nicht nur bestehende Anforderungen an Software, sondern es entstehen auch neue Kriterien. Ganz abgesehen davon steigt die Komplexität von modernen Software-Systemen immens an.

Mit steigender Komplexität und höherer Nachfrage am Markt, sowie engen Zeitplänen für Projekte wird leider häufig aus Zeit- und Kostengründen auf Qualität nur geringfügig Rücksicht genommen. Zunächst verursacht eine gute Software-Qualität nämlich Mehrkosten. Personelle wie zeitliche. Dies zeigt das Magische Dreieck, oder im englischen das Project Management Triangle.

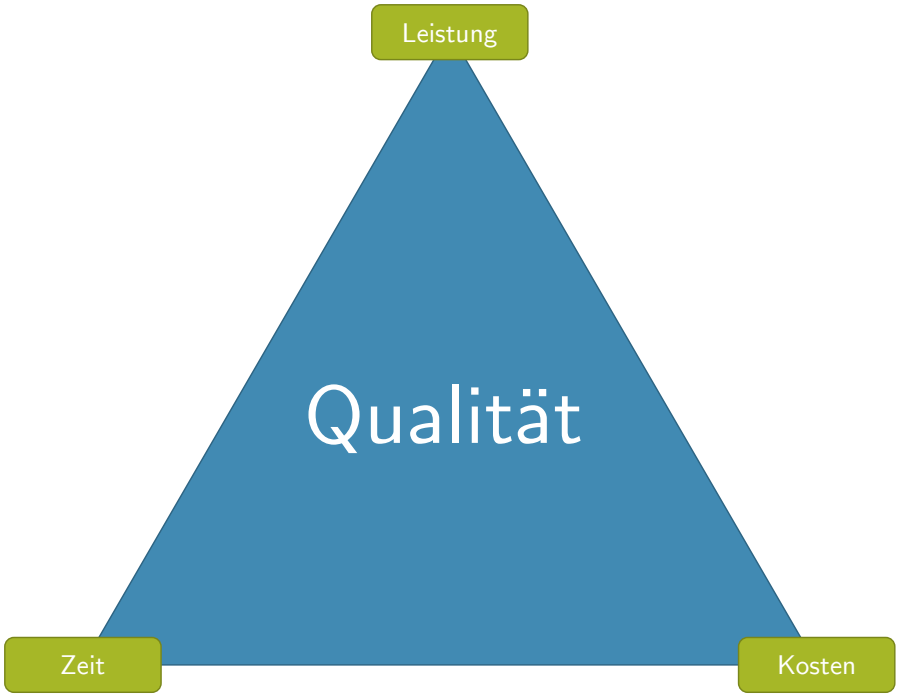

Abbildung 1: Magisches Dreieck des Projektmanagements[Hag08]

Dieses besagt, dass die Qualität eines Projekts durch die drei Faktoren der Leistung, Kosten und Zeit beeinflusst wird. Ändert sich einer dieser Faktoren, ändern sich jedoch auch alle anderen mit.

Ein Bericht der Kölner Beratungsfirma SQS zeigt anhand von gesammelten Zahlen aus Beratungsaufträgen welche immensen Kosten durch unentdeckte Fehler entstehen. Hier

wird besonders deutlich wie viel es für ein Projekt bedeutet, frühzeitige Qualitätssicherung durchzusetzen. Und dazu zählt auch das Testen von Software.

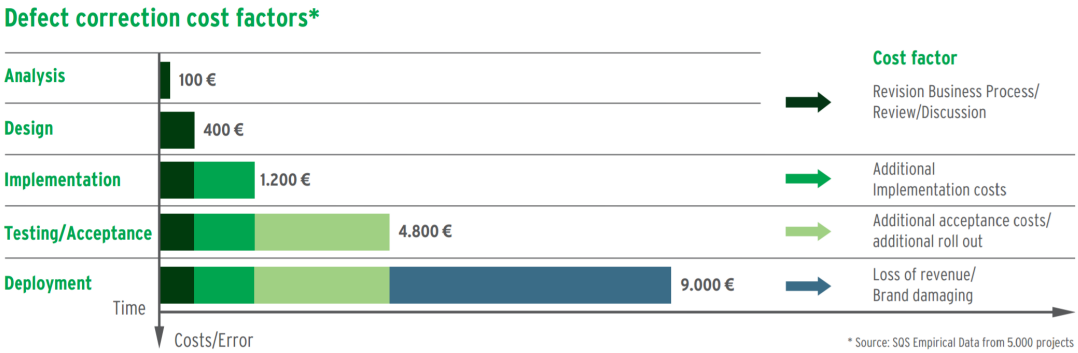

Abbildung 2: SQS Report Costs of Defect Correction [SQS]

Umso früher Fehler entdeckt und bemerkt werden, umso weniger kostet es auch diese zu beheben. Wenn bereits vor dem Start der Implementierungsphase auf eine hohe Testabdeckung, beispielsweise durch den Einsatz von test-driven development, wert gelegt wird, können, je auftretendem Fehler, um die 7800€ [SQS] eingespart werden. Mit diesen Zahlen sind die Mehrkosten, die für ein solches vorgehen entstehen um ein vielfaches leichter zu rechtfertigen.

Somit sorgt das Bug-Fixing in Produktivsystemen, also das Beheben sogenannter *field defects*, für einen der größten Kostenfaktoren. Wurde in den ersten Phasen eines Projekts nicht viel, oder kein Wert auf eine ausreichende Test-Abdeckung gelegt schaffen es viele Fehler in die Produktivsysteme der Hersteller. Doch werden diese Fehler erst im laufenden Betrieb beim Kunden festgestellt, ist es bereits zu spät. Robert N. Charette kritisiert eben dies in seinem Artikel *Why Software Fails*[Cha05].

> *„If the software coders don't catch their omission until final system testing–or worse, until after the system has been rolled out–the costs incurred to correct the error will likely be many times greater than if they'd caught the mistake while they were still working on the initial [...] process.“*

Die Lösung sollte also sein, viel Zeit und Geld in gute Softwarequalität zu investieren. Jedoch stehen, wie bereits am Anfang der Einleitung erwähnt, Projektleiter und ihre Mitarbeiter unter hohem zeitlichen Druck vom Kunden oder durch andere beteiligte festgelegte Termine einzuhalten. Und gute Tests kosten neben Geld auch Zeit. Es entsteht in dieser Zeit aber kein Fortschritt an der Funktionalität der Software.

Bei it@M, dem IT-Dienstleister der Landeshauptstadt München, ist eine hohe Testabdeckung daher Teil der Definition of Done [citation needed] von Softwareprojekten. Im

Kommunalen Umfeld sind die zeitlichen Restriktionen noch einmal stärker zu gewichten als in der freien Wirtschaft. Viele Projekte werden aufgrund von anstehenden Gesetzesänderungen ins Leben gerufen und müssen mit Inkrafttreten der neuen Regelungen in Produktion gehen. it@M ist somit ständig auf der Suche nach Lösungen, die den Entwicklungs- und Testprozess beschleunigen, um die geringe Zeit möglichst effizient nutzen zu können.

Eine dieser Lösungen wurde im letzten Jahr von Martin Kurz im Rahmen seiner Masterarbeit [Kur16] geplant und entwickelt. Die Model-driven Software Development Lösung Barrakuda [1]. Diese bietet den Entwicklern von it@M die Möglichkeit anhand von einer vorgegebenen Domänen-spezifischen Sprache Microservice-Architekturen zu modellieren und diese zu generieren.

Im Rahmen dieser Arbeit soll eine Erweiterung von Barrakuda geplant und entwickelt werden. Diese Weiterentwicklung soll einen Großteil verschiedener Testmethoden für diese Architektur generieren und die benötigte Entwicklungszeit für eine hohe Testabdeckung möglichst stark reduzieren.

Zunächst werden die Microservice Architektur (Kapitel 2.1) und der Monolithischen Architekturstil (Kapitel 2.2) beschrieben und bezüglich ihrer Vor- (Kapitel 2.3) und Nachteile (Kapitel 2.4) verglichen. Anschließend wird ein praktisches Beispiel anhand der von it@M Vorgegebene Microservice-Architektur gezeigt und die zu testenden Komponenten identifiziert.

Schließlich werden bekannte Methoden zum Testen von Software analysiert (Kapitel 3.1) und es werden alternative Methoden, die besonders im Bereich der Microservices anzutreffen sind, ebenfalls untersucht (Kapitel 3.2). Diese werden dann in Hinblick der Umsetzungsmöglichkeit im generativen Ansatz geprüft (Kapitel 3.2.1) und es werden Frameworks die zur Implementierung genutzt werden sollen definiert(Kapitel 3.2.2) .

Schließlich sollen Anforderungen, die an die zu generierenden Tests gestellt werden (Kapitel 4), festgehalten werden. Diese finden dann im letzten Abschnitt, der Implementierung, Beachtung (Kapitel 5).

2 Architekturvergleich Microservice und Monolith

Architekturen gibt es in der Welt der Softwareentwicklung viele. Eine der momentan bekanntesten Neuerungen an den Whiteboards der IT-Architekten ist die "MicroserviceÄrchitektur. Viele kleine Services die im Konglomerat für ein gemeinsames Ziel zu-

[1] Github Repository von Barrakuda (https://github.com/xdoo/mdsd)

sammenarbeiten. Vorbei sein soll die Zeit der Monolithischen Softwaresysteme, bei denen die Gesamtlogik in einer einzigen Anwendung gekapselt ist.

Sam Newman stellt in seinem Buch *Building Microservices* einige Vorteile dar, die Microservices gegenüber Monolithischen Architekturen bieten. Doch um diese Vorteile besser zu verstehen muss man zunächst die Unterschiede der zwei Architekturstile im Detail betrachten.

2.1 Microservice Architektur

Um die Vorteile dieser Architektur in einem System effektiv zu nutzen gibt es bereits in der Planung einiges zu beachten.

Dazu zählt, dass es zwei Dinge gibt, die einen guten Microservice ausmachen. Lose Kopplung und Starker Zusammenhalt [New15, S.62].

Lose Kopplung bedeutet, dass Änderungen an einem Service keine Änderungen an einem anderen Service nach sich ziehen. Ist dies nicht gegeben, ist einer der Hauptvorteile von dieser Art Architektur nicht mehr vorhanden. Ein lose gekoppelter Service weiß von seinem Kommunikationspartner nur so wenig wie möglich.[New15, S.63]

Starker Zusammenhalt soll dafür sorgen, dass bestimmte Funktionalitäten an einem Ort vorhanden sind, sodass diese leicht geändert werden können und nicht mehrere Komponenten angepasst werden müssen.[New15, S.64]

Zum Planen einer Software mit Mircoservice Architektur ist es sinnvoll, sich zunächst über Kontextgrenzen (*bounded context*) Gedanken zu machen. Kontextgrenzen sind eine Definition aus dem Buch *Domain Driven Design* von Eric Evans [Eva03].

Sie lassen sich gut am Beispiel eines Online-Shops zeigen. Wenn ein Online-Shop geplant wird ist einer der zentralen Bestandteile die Lagerverwaltung. Die Daten der Lagerverwaltung werden herangezogen um dem potentiellen Kunden des Shops anzuzeigen welche und wie viele Produkte verfügbar sind. Doch möchte ein Kunde nun ein Produkt bestellen hat dies nichts mehr mit einer Lagerverwaltung, sondern mit einem Bestellsystem zu tun. Man verlässt also den ursprünglichen Kontext der Lagerverwaltung. Somit entsteht in der Planung eine neue Kontextgrenze: Die des Bestellsystems. Es verwaltet Bestellungen von Kunden. Doch woher kommen die Kundendaten? Kundendaten haben zwar einen Verwendungszweck im Bestellsystem, doch die Verwaltung der Daten hat mit der eigentlichen Kompetenz dieses Systems nichts mehr zu tun. Somit wird eine weitere Kontextgrenze erstellt, nämlich die der Kundenverwaltung.

Eine Kontextgrenze soll also eine logische Grenze darstellen, welche eventuell über ei-

ne oder mehrere Schnittstellen verfügt, die festlegen, welche Informationen mit anderen Kontexten geteilt werden.[New15, S.65]. Damit einher geht die Unterscheidung zwischen geteilten und versteckten Modellen. Versteckte Modelle werden innerhalb einer Kontextgrenze benötigt, sind aber für andere Kontexte uninteressant. Geteilte Modelle hingegen werden über die Grenzen hinweg freigegeben. Sind solche Kontextgrenzen für eine Software modelliert, lassen sich aus diesen sehr leicht Microservices ableiten, da bereits einige Grundvoraussetzungen getroffen sind: Lose Kopplung und Starker Zusammenhalt.[New15, S.68]

> *„[I]f our service boundaries align to the bounded contexts in our domain, and our microservices represent those bounded contexts, we are off to an excellent start in ensuring that our microservices are loosely coupled and strongly cohesive.[New15]“*

2.1.1 Kommunikation zwischen Services

Als weiterer Schritt muss eine Kommunikationsart zwischen Services und zur Außenwelt definiert werden.

2.1.1.1 Geteilte Datenbanken sind eine Möglichkeit zur Servicekommunikation [New15, S.85]. Die Idee hinter dieser Art der Kommunikation ist besonders trivial. Mehrere Services, die miteinander Daten austauschen wollen, nutzen eine gemeinsame Datenbank. Jeder Service hat somit jederzeit Zugriff zu allen Daten der anderen Nutzer dieser geteilten Datenbank und es findet somit keine *echte* Kommunikation statt, sondern nur Zugriffe auf geteilten Speicher.[Hoh04] Dies birgt natürlich viele Probleme. Jede noch so kleine Änderung an der "Logik"dieser Datenbank, oder an der internen Struktur der Daten muss mit viel Bedacht durchgeführt werden, da jede abhängige Komponente sonst nicht mehr funktionieren könnte.

Des Weiteren ist die technologische Einschränkung ein großer Nachteil. Wenn es auch in den ersten Schritten der Planung und Entwicklung Sinnvoll erscheint eine relationale Datenbank zu verwenden, können spätere Geschäftsentscheidungen oder neu auftretende Probleme den Einsatz einer Graphdatenbank sinnvoller machen. Bei einer geteilten Datenbank eine solche Änderung durchzuführen ist sehr schwierig[New15, S.85].

2.1.1.2 Remote Procedure Call (RPC)s sind eine weitere bekannte Kommunikationsmöglichkeit. Die Übersetzung ins deutsche erklärt schon einen Großteil der Idee hinter RPCs: Äufruf einer fernen Prozedur". RPCs sind eine weitere Möglichkeit eine Client-Server-Modell umzusetzen. Die erste Idee dazu kam im Jahre 1976 von James White in sei-

nem RFC #707 „A High-Level Framework for Network-Based Resource Sharing“[Whi76]. Ein RPC funktioniert, indem der Client einer Anwendung eine Anfrage an einen Server sendet. In dieser Anfrage ist der Name oder die ID einer Methode, sowie die zugehörigen Parameter enthalten. Der empfangende Server führt die gewünschte Methode bei sich aus und liefert dem Client als Antwort den Rückgabewert der Methode. Ein Vorteil von RPC ist, dass es sehr einfach und schnell möglich ist Methoden der Services für Clients und andere Services Verfügbar zu machen. Über die Kommunikation muss man sich nahezu keine Gedanken machen[New15, S.91].

Doch die Nachteile überwiegen schnell und deutlich. Je nach Implementierung führt die Verwendung von RPC zu einer starken Bindung an eine bestimmte Sprache (bekanntestes Beispiel Java RMI). Auch verstecken RPC-Implementierungen die Komplexität der entfernten Aufrufe. Dies kann zu starken Performance-Problemen führen, wenn Entwickler an Interfaces arbeiten, von denen sie denken es seien lokale Methoden[New15, S.93].

Die Verwendung von RPC macht die Weiterentwicklung von Systemen nicht einfacher: Ein Beispiel anhand einer Schnittstelle zur Instanziierung eines Kunden. Zusätzlich zur Erstellung eines Kunden mithilfe seines Namen und einer E-Mail Adresse soll es nun eine Möglichkeit geben diesen nur mithilfe seiner Mail-Adresse zu erstellen. Das reine hinzufügen einer neuen Methode in einem Interface löst das Problem in diesem Fall nicht. Im schlimmsten Fall benötigen alle Clients die diesen Service ansprechen die neuen Stubs und müssen allesamt neu Bereitgestellt werden[New15, S.94]. Und das bereits bei einer so kleinen Änderung.

2.1.1.3 Representational State Transfer (REST) ist eine der meistgenutzten Programmierparadigmen für APIs [DuV10]. Entworfen wurde REST von Roy Fielding, der die Idee und Spezifikationen in seiner Dissertation „Architectural Styles and the Design of Network-based Software Architectures“ veröffentlichte.[Fie00]. Insgesamt besteht Fielding auf 6 Eigenschaften, die erfüllt sein müssen um eine REST-Konforme Anwendung zu schreiben.

Die erste dieser Eigenschaften („Client-Server“) verlangt schlichtweg, dass eine Client-Server Architektur vorliegt, bei der ein Server Funktionalität bereitstellt, die der Client nutzen kann. „Stateless“ ist die zweite Eigenschaft, die von Fielding gefordert wird. Sie besagt, dass alle Informationen, die zum Verständnis einer Nachricht nötig sind auch mitgeliefert werden müssen, da weder der Client, noch der Server, Zustandsinformationen zwischen Anfragen speichern sollen. Eigenschaft Nummer 3, „Caching“, fordert die Verwendung von Caching um Netzwerklast zu minimieren. Das „Uniform Interface“ ist

eine der größten Unterschiede von REST zu anderen Netzwerkarchitekturstilen. Diese Eigenschaft ist erneut aufgeteilt in vier Unterpunkte:

Identification of resources Jede, über eine URI erreichbare Information ist eine Ressource

Manipulation of resources through representations Änderungen an Ressourcen erfolgen nur über Repräsentationen von Ressourcen, also zum Beispiel durch Formate wie XML oder JSON. Genauso gut kann eine Ressource aber auch in unterschiedlichen Darstellungsformen vom Server ausgeliefert werden.

Self-descriptive messages Die für die Anwendung versendeten Nachrichten sollen selbstbeschreibend sein. Zur Erfüllung dieser Eigenschaft ist es unter anderem nötigt, Standardmethoden zur Manipulierung von Ressourcen zu verwenden.

Hypermedia as the engine of application state (HATEOAS) ist ein wichtiger Teil der REST-Spezifizierung und beschreibt ein Konzept, nach dem, einfach gesagt, Informationen Links zu anderen Informationen enthalten.

Die fünfte Eigenschaft verlangt nach „Layered Systems“, also mehrschichtigen Systemen. Die Idee dahinter ist, dass die Komponenten nur die Komponenten im System kennen mit denen sie in direkter Interaktion stehen. Alles weitere bleibt verborgen. „Code-On-Demand“ ist die letzte und eine optionale Eigenschaft der Spezifikation. Fielding beschreibt hiermit die Erweiterung von Client-Funktionalität durch den Server. Dieser sendet, wie zum Beispiel mit Javascript der Fall, Code für den Client direkt an den Client,.[Fie00]

REST wird, auch wenn die Spezifikation es nicht vorschreibt, in den häufigsten Fällen über HTTP genutzt [New15, S.97]. Dies rührt daher, dass beispielsweise die bekannten HTTP-Methoden POST, GET, PUT usw. es sehr einfach machen die geforderte homogene Verhaltensweise von Methoden auf allen Ressourcen umzusetzen. Auch wird HTTP gerne genutzt, da es bereits eine breite Masse an bestehenden Tools gibt die zur weiteren Qualitätsverbesserung eines Systems genutzt werden können, wie zum Beispiel Proxies und Load Balancer. Doch dies sind alles zunächst nur Vorteile von HTTP.

Die Verwendung von REST bietet viele Möglichkeiten die lose Kopplung zwischen Services zu ermöglichen. Dazu zählt unter anderem HATEOAS. Um bei dem Kundenbeispiel aus 2.1.1.2 zu bleiben: Wird die Information eines Kunden abgerufen, kann das Kundenobjekt zusätzlich zu den Kundendaten ein Feld enthalten welches zur Bestellliste dieses Kund zeigt.

```
1 {
```

```
2     "name": "Hans Meier",
3     "mail": "hansmei@mail.de",
4     "links": [{
5       "rel": "orders",
6       "href": "https://mycompany.com/orders/12341232/"
7     }]
8   }
```

Mit der Verwendung von HATEOAS reicht es, wenn alle Clients die Kundendaten und deren Bestellungen verarbeiten wissen, dass Kunden einen Link-Feld mit dem Typ *orders* besitzen. Wenn also später Services unter anderen Adressen erreichbar sind, oder sich interne Datenstrukturen ändern müssen diese Clients nicht neu angepasst werden.

2.1.1.4 JSON oder XML?

Wenn die Entscheidung über die Art und Weise der Datenübertragung gefallen ist muss das Datenformat festgelegt werden. JSON und XML sind dabei die bekanntesten Namen. In den vergangenen Jahren ist dabei die Verwendung von XML im Gegensatz zu JSON in APIs zurückgegangen[DuV13]. Jedoch bieten beide Vor- wie Nachteile. JSON ist das einfachere Format und auch leichtgewichtiger, während XML einige sinnvolle Features wie z.B. link control (insbesondere für HATEOAS interessant) oder auch das extrahieren von Teilinformationen durch Standards wie in etwa XPATH[New15, S.101].

2.1.1.5 Authentifizierung

todo

2.2 Monolithische Architektur

Monolithische Software stellt das Gegenteil zu Microservices dar. Eine Monolithische Anwendung vereint die Gesamtlogik in einer einzigen Laufzeitumgebung und ist dadurch meist sehr groß. Die Aufteilung der Logik innerhalb der Anwendung findet nur auf Entwickler-Ebene statt. So zum Beispiel durch Packages in Java, oder Libraries die unabhängig voneinander kompiliert und entwickelt werden können, aber dennoch am Ende eine große Anwendung bilden.

Die Probleme bei monolithischen Systemen treten nicht zu Beginn der Entwicklung eines Produkts auf. Am Anfang stehen sogar einige Vorteile und gute Gründe, warum ein solcher Ansatz sinnvoll sein kann. Die Entwicklung ist sehr einfach. Entwicklungstools und IDEs sind erst in der letzten Zeit dazu übergegangen, auch die Entwicklung von Microservices besser zu unterstützen, während der Support für monolithische Architekturen bereits vorhanden ist, da dies auch der bisherige Ansatz zur Entwicklung war. Das Deployment

ist weitaus einfacher als die Orchestrierung, die für Microservices nötig ist. Beispielsweise reicht es bereits, ein einfaches WAR-File auf einem Application-Server auszuliefern. Auch die Skalierung ist möglich, durch die Verwendung eines Load-Balancers und das starten von mehreren Instanzen der Anwendung.[Ric14]

Doch ab einem gewissen Entwicklungsstand nehmen die Probleme überhand. Viele Leute arbeiten an einer Code-Basis die stetig wächst. Wenn über den Projektzeitraum neue Entwickler am Projekt teilnehmen können diese zunächst leicht überfordert sein, da die anfängliche Modularität dadurch, dass sie nicht hart vorgeschrieben ist, stetig abnimmt. Dies führt auch dazu, dass bei Änderungen an großen Systemen nicht mehr direkt klar ist, welche Auswirkungen es auf andere Teilkomponenten und somit das Gesamtsystem gibt. Die Entwicklungsgeschwindigkeit verringert sich.

Auch die IDE trägt dazu bei. Je größer, das Projekt, umso höher die Belastung für den Entwicklungsrechner. IDEs indizieren Source-Files um die Navigation zu vereinfachen. Je nach Projektgröße kann so der Arbeitsspeicher schnell ausgelastet sein.

Eine größer werdende Anwendung leidet auch unter dem zunächst einfachen Deployment-Prozess. Riesige Java-Anwendungen brauchen lange Zeit zum starten, was auch für Integrationstests zu hohen Zeiteinbußen führt.

Kleine Änderungen in Produktion zu geben ist eine große Aufgabe. Immer muss das Gesamtsystem neu gestartet werden, auch wenn nur ein prozentual kleiner Anteil der Anwendung umgeschrieben wird. Release-Zyklen verlangsamen sich auf das nötigste. Fehler bleiben eventuell länger im System.

Auch wenn die Skalierung in der Theorie sehr einfach funktionieren sollte, tut sie das in der Praxis meist nicht. Das Problem: Monolithen lassen sich nur eindimensional Skalieren. Auf eine höhere Anzahl von Anfragen kann also durch das starten einer weiteren Instanz reagiert werden. Doch wenn Teilkomponenten von Überlast betroffen sind wird es ineffizient. Beispielsweise gibt es eine Teilkomponente die sehr CPU-Intensive Berechnungen durchführt. Es müssen weitere Instanzen gestartet und dadurch Ressourcen belastet werden, die von der CPU-Intensiven Komponente besser verwendet werden könnten.

Der letzte große Nachteil tritt häufig erst lange nach Entwicklungsbeginn auf. Die Anwendung ist gewachsen und neue Features werden verlangt, doch die zu Beginn eingesetzte Sprache ist für bestimmte Einsatzzwecke nur schlecht zu gebrauchen. Doch man ist durch den monolithischen Ansatz gebunden. Auch kann es passieren, dass die Aktualisierung auf neuere Sprachversionen nicht mehr, oder nur schwer möglich wird. Alles durch die größe der Applikation. Im schlimmsten Fall wird der Support für die aktuell verwendete Sprache eingestellt.[Ric14]

2.3 Vorteile von Microservices gegenüber Monolithen

Als Vorteile der Microservice-Architektur lassen sich nun folgende Punkte, die Newman in seinem Buch nahelegt bestätigen[New15]:

Die Heterogenität, welche es erlaubt, für verschiedene Einsatzzwecke verschiedene Sprachen und Technologien zu verwenden, ohne das ganze System damit implementieren zu müssen.

Die erhöhte Widerstandsfähigkeit gegenüber Fehlern im System, da die Grenzen von Microservices eine Kaskadierung von Fehlern verhindern können.

Microservices lassen sich leichter skalieren. Während monolithische Architekturen immer im ganzen skaliert werden, kann man in einer Microservice-Architektur nach genau die Services skalieren, die in diesem Moment mehr Leistung benötigen.

Ein weiterer Punkt ist ein einfacheres Deployment. Insbesondere kleinen Änderungen führen bei monolithischen Systemen zu einem großen Overhead, während man in einer Microservice Architektur nur den Service neu deployen muss, der auch die implementierte Änderung enthält.

Ebenfalls lässt sich die Team-Organisation vereinfachen. Ein Team arbeitet an einem Service, dessen Funktionalität klar definiert ist, anstatt ein großes Team zu haben, wessen Teilteams an teilen eines Monolithen arbeiten.

Auch optimieren Microservices die Austauschbarkeit. Die kleinen abgegrenzten Systeme lassen sich mit viel weniger aufwand gegen neuere oder bessere Implementierungen austauschen, ohne andere Komponenten zu Gefährden. Während dies bei Monolithischen Anwendungen zu unvorhersehbaren Problemen kommen kann, weshalb in solchen Architekturen auch häufig kaum Änderungen durchgeführt werden.

Dies sind einige der Gründe, warum it@M nun mehr auf die Microservice Architektur setzen möchte. Sie behandelt einen Großteil der Probleme, die bei bestehenden Systemen der Landeshauptstadt in der Vergangenheit aufgetreten sind.

2.4 Nachteile von Microservices gegenüber Monolithen

2.5 Vorgegebene Architektur der Landeshauptstadt München (LHM)

Die Entscheidungen, welche Technologien und Architekturen verwendet werden sind bereits von it@M getroffen worden. Die vorgegebene Microservice-Architektur verwendet das Spring-Framework. Auslöser dafür ist, dass bei it@M größtenteils Java-Entwickler

arbeiten und diese bereits mit Java EE viel Erfahrung gesammelt haben. Da das Spring-Framework auf viele Konzepte von Java EE aufbaut, erleichtert dies den Einstieg in die Microservice-Welt für kommende Entwicklungen.

Die Services kommunizieren über eine REST-API, die ihre Daten im JSON-Format überträgt. HATEOAS wird über Spring HATEOAS [2] genutzt.

Als Datenbank in der Entwicklungsumgebung kommt eine In-Memory-Datenbank von H2 zum Einsatz. In der Produktion größtenteils Oracles JDBC-Datenbank in der Version 7, aber auch MySQL wird verwendet.

Zur Authentifizierung wird ein Authentifizierungs-Service genutzt, der zusätzlich zur internen Nutzer- und Rechteverwaltung auch eine Anbindung an das Stadtweite Lightweight Directory Access Protocol (LDAP) bietet.

Das Problem der Findung von Service wird mithilfe von Netflixs Eureka-Service gelöst. Dieser bietet innerhalb einer Domäne einen Zentralen Anlaufpunkt für alle Services um sich dort zu vermerken, sowie Informationen über die Adressen von anderen Services einzuholen.

Clients können mit der Domäne über ein API-Gateway kommunizieren (Netflix Zuul). Dieses öffnet nach außen eine einzige Schnittstelle, über die sowohl eine graphische Nutzeroberfläche aufgerufen, als auch die einzelnen Services kontaktiert werden können. Auch ermöglicht das Gateway den Zugriff auf bestimmte Endpunkte zur sperren, was für die Landeshauptstadt insbesondere für Software interessant ist, die sowohl interne, als auch externe Schnittstellen bieten soll.

Auch werden Docker-Konfigurationen verwendet, um die Continous Delivery mithilfe eines Container-Frameworks zu ermöglichen.

[2] `http://projects.spring.io/spring-hateoas/`

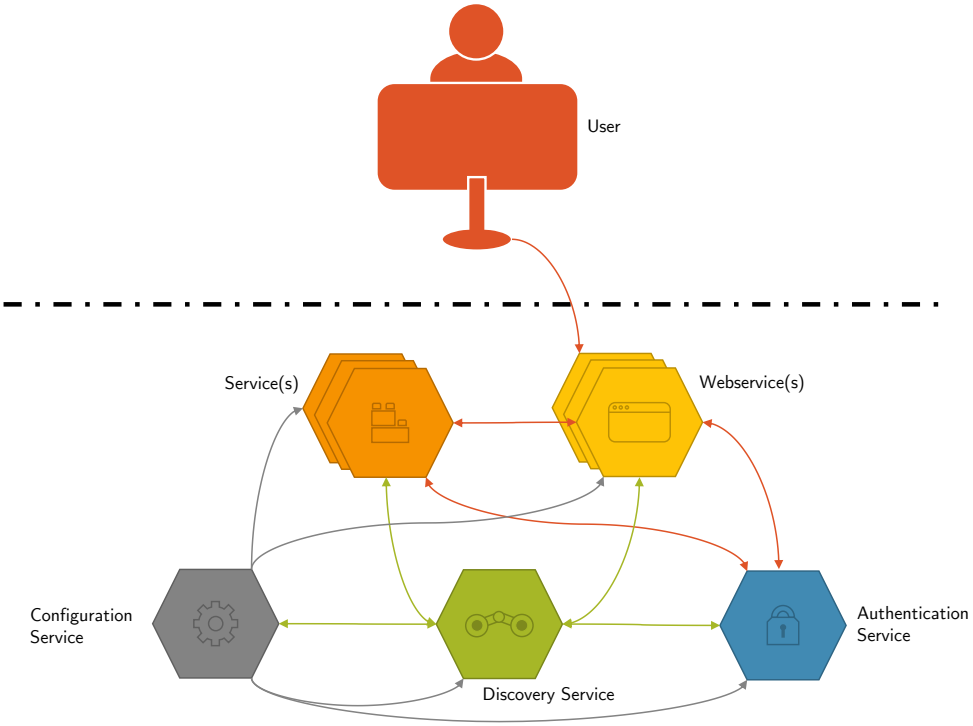

Abbildung 3: Architekturvorgabe it@M

Intern sehen die generierten komponenten so aus:

2.5.1 Authentifizierungsservice

2.5.2 Discoveryservice

2.5.3 Configurationservice

2.5.4 Service

2.5.5 Webservice

3 Software Testen

3.1 Bekannte Testmethoden

3.2 Testen von Microservices

3.2.0.1 Unit Tests

3.2.0.2 Integrationstests

3.2.0.3 Komponententests

3.2.0.4 Contract Testing

3.2.0.5 End-To-End Testing

3.2.1 Sinnvolle Teststrategien für den generativen Ansatz

3.2.2 Frameworks zum Umsetzen von Test-Strategien

4 Anforderungen an generierte Tests

4.1 Benötigte Daten

4.2 Notwendige Änderungen/Erweiterungen von Barrakuda

5 Implementierung in Barrakuda

5.1 Referenz-System

Als Referenz-System wird ein vereinfachtes Modell eines Online-Shops, beispielhaft *Kongo* genannt, herangezogen. Das Model in der Barrakuda-Sprache sieht folgendermaßen aus:

```
1  domain kongo package edu.hm.ba version 1.0;
2
3  serviceModel shoppingcart package edu.hm.ba.kongo.shop version 1.0 {
4
5      customNumberType intMin1 minValue=1;
6
7      entity Cart {
8          items manyToMany CartItem;
9          userID warehouse.OID "123";
10         totalPrice warehouse.float "15.5";
11     }
12
13     valueObject CartItem {
14         product warehouse.OID mainFeature "123";
15         quantity intMin1 mainFeature "1";
16     }
17
18     businessAction addToCart {
19         purpose "Add a product from the warehouse to the users current
               shopping cart";
20         given productID warehouse.OID;
21         given quantity intMin1;
22     }
23
24 }
```

```
25
26 serviceModel ordering package edu.hm.ba.kongo.shop version 1.0{
27
28     customTimeType dateInPast inThePast;
29
30     entity order {
31         cart warehouse.OID "123";
32         orderedOn dateInPast "10.10.2010";
33     }
34
35     businessAction orderCart {
36       purpose "Receives a shopping cart to create a new order which can
             then be payed";
37       given cartID warehouse.OID;
38   }
39
40     businessAction sendInvoice{
41         purpose "Sends the value of the costs of the ordered procuts to
             an invoicing system";
42         given orderID warehouse.OID;
43     }
44
45     businessAction cancelOrder{
46         purpose "Deletes an order and the associated shopping cart with
             it's contents";
47         given orderID warehouse.OID;
48     }
49
50 }
51
52 serviceModel warehouse package edu.hm.ba.kongo.shop version 1.0{
53
54     customTextType OID maxLength=24;
55     customTextType stringMin1 minLength=1;
56     customTextType longText type=long;
57   customNumberType float pointNumber minValue=0;
58   customNumberType int minValue=0;
59
60     entity Product {
61         name stringMin1 searchable mainFeature "Bottle";
62         description longText optional "Great for storing water";
63         price float mainFeature "10.5";
64         quantity int mainFeature "5";
65     }
66
```

```
67 }
```

Das Referenz-System besteht aus 3 Services, dem *shoppingcart*-Service, dem *ordering*-Service und dem *warehouse*-Service. Im *warehouse* werden alle verfügbaren Produkte des Online-Shops verwaltet. Wenn ein Kunde nun eines der Produkte bestellen möchte, wird eine Anfrage an den [shoppingcart]-Service gesendet, die die OID des Produktes, sowie die gewünschte Anzahl enthält.

Wenn ein Kunde dann eine Bestellung aufgeben möchte, geht eine Anfrage mit der OID des virtuellen Einkaufswagens an den *ordering*-Service, der zusätzlich Schnittstellen zum erstellen einer Rechnung, sowie zum stornieren einer Bestellung bietet.

Der mit generierte Authentifizierungsservice dient als Kundenverwaltung.

Wichtig ist, dass im Referenz-System keinerlei Logik zu den modellierten Geschäftsanwendungen implementiert wird, sondern lediglich die im Abschnitt 3.2 angesprochenen Testmethoden.

5.1.1 Komponenten und Aufbau

5.1.2 Implementierung des Systems

5.1.3 Implementierung der Tests

5.2 Übernahme der Referenz-Implementierung in Barrakuda-Templates

6 Fazit

7 Quellenverzeichnis

[Cha05] Charette, Robert N.: Why Software Fails. (2005). http://spectrum.ieee.org/computing/software/why-software-fails

[DuV10] DuVander, Adam: New Job Requirement: Experience Building RESTful APIs. (2010). http://www.programmableweb.com/news/new-job-requirement-experience-building-restful-apis/2010/06/09. – Zuletzt Abgerufen am 12.12.2016

[DuV13] DuVander, Adam: JSON's Eight Year Convergence With XML. (2013). http://www.programmableweb.com/news/jsons-eight-year-convergence-xml/2013/12/26. – Zuletzt Abgerufen am 12.12.2016

[Eva03] Evans, Eric: *Domain-driven Design*. Addison-Wesley Professional, 2003. – ISBN 9780321125217

[Fie00] Fielding, Roy T.: Architectural Styles and the Design of Network-based Software Architectures. (2000). http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. – Zuletzt Abgerufen am 13.01.2017

[Hag08] Hagen, Stefan: Das "Triple Constraintïm Projektmanagement. (2008). http://pm-blog.com/2008/08/04/triple_constraint_magisches_dreiec/. – Zuletzt Abgerufen am 10.01.2017

[Hoh04] Hohpe, Gregor: Enterprise Integration Styles. (2004). http://www.informit.com/articles/article.aspx?p=169483&seqNum=3. – Zuletzt Abgerufen am 13.01.2017

[Kur16] Kurz, Martin: *Verbesserung des Softwareentwicklungsprozesses der Landeshauptstadt Muenchen durch modellgetriebene Softwareentwicklung*, Hochschule München, Diplomarbeit, 2016

[New15] Newman, Sam: *Building Microservices*. O'Reilly, 2015. – ISBN 9781491950357

[Ric14] Richardson, Chris: Pattern: Monolithic Architecture. (2014). http://microservices.io/patterns/monolithic.html. – Zuletzt Abgerufen am 16.01..2017

[SQS] SQS: Detect errors early on, reduce costs and increase quality. https://www.sqs.com/en/academy/download/fact-sheet-EED-en.pdf

[Whi76] White, James E.: A High-Level Framework for Network-Based Resource Sharing. (1976). https://tools.ietf.org/html/rfc707. – Zuletzt Abgerufen am 24.01.2017

Anhang

A Code-Fragmente

Viel Beispiel-Code