



Fakultät für Informatik und Mathematik 07

Bachelorarbeit

über das Thema

Generative Testerstellung für Microservice-Architekturen

Autor: Fabian Holtkötter
holtkoet@hm.edu

Prüfer: Prof. Dr. Ulrike Hammerschall

Abgabedatum: 03.03.2017



Diese Erklärung ist zusammen mit der Bachelorarbeit bei der PrüferIn abzugeben.

(Familienname, Vorname)

(Ort, Datum)

(Geburtsdatum)

 / 20

(Studiengruppe / WS/SS)

Erklärung

Hiermit erkläre ich, dass ich die Bachelorarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

(Unterschrift)

I Kurzfassung

Ziel dieser Arbeit ist es, durch generative Testerstellung die Qualität von Microservice-Anwendungen zu steigern, ohne dabei Kosten und Zeitaufwand des Projekts zu erhöhen. Dazu werden zunächst spezifische Vorgehensweisen zum Testen von Microservice-Architekturen erläutert. Im nächsten Schritt werden diese dann in einem Referenzsystem implementiert. Bei der Implementierung wird Rücksicht auf die Generierbarkeit aller Tests genommen, sodass diese in die bestehende Infrastruktur zur Generierung von Microservice-Architekturen integriert werden können. Durch die Arbeit hat sich gezeigt, dass der generative Ansatz teilweise die angestrebte Qualitätssteigerung ermöglicht, wenn auch einige Einschränkungen in der Bandbreite der generierten Tests zu erkennen ist.

Abstract

This thesis aims to enhance the quality of a application in microservice-architecture without increasing time and cost expenses of the project by generating tests. To validate this claim specific methods are explained which are especially useful for testing microservice architectures. These methods then get implemented while taking into concern that these tests shall be generated at a later point. This thesis has shown that the generation of these tests partly fulfills the desired rise in quality even though there are recognizable limitations to the spectrum in which the generated tests can operate.

II Inhaltsverzeichnis

I	Kurzfassung	I
II	Inhaltsverzeichnis	II
III	Abbildungsverzeichnis	III
IV	Tabellenverzeichnis	III
V	Listing-Verzeichnis	III
VI	Abkürzungsverzeichnis	IV
1	Einführung und Motivation	1
2	Architekturvergleich Microservice und Monolith	4
2.1	Microservice Architektur	5
2.1.1	Definition von Service-Grenzen	5
2.1.2	Kommunikation zwischen Services	7
2.1.3	Datenformat - JSON oder XML?	15
2.1.4	Autorisierung mit OAuth2	15
2.1.5	Deployment	17
2.2	Monolithische Architektur	18
2.3	Vorteile von Microservices gegenüber Monolithen	18
2.4	Nachteile von Microservices gegenüber Monolithen	22
2.5	Resümee des Vergleichs	22
3	Vorgegebene Architektur der Landeshauptstadt München (LHM)	23
3.1	Backing Services der Architektur	23
3.2	Kommunikation zwischen den Komponenten	24
3.3	Interne Struktur der Services	25
4	Software Testen	27
4.1	Bekannte Testmethoden	27
5	Konzeption: Testen von Microservices und generative Testerstellung	29
5.1	Unit Testing	29
5.2	Integration Testing	30
5.3	Component Testing	31
5.4	Consumer Driven Contract Testing (CDCT)	32
5.5	End-To-End Testing	35
5.6	Code-Generierung	38
6	Implementierung	39
6.1	Referenz-System	39
6.1.1	Architektur	39
6.1.2	Beispiel-Anwendung	40

6.2	Frameworks zum Umsetzen von Test-Strategien	42
6.3	Generativer Ansatz - Was zu beachten ist	43
7	Fazit	43
8	Quellenverzeichnis	45

III Abbildungsverzeichnis

Abb. 1	Magisches Dreieck des Projektmanagements [Hag08]	1
Abb. 2	SQS Report Costs of Defect Correction [SQS]	2
Abb. 3	Logische Unterscheidung zwischen monolithischer und Microservice Architektur	4
Abb. 4	Beispiel der Kontextgrenzen des Online-Shops	6
Abb. 5	Aggregates im Online-Shop Beispiel	7
Abb. 6	Ablauf eines RPC	9
Abb. 7	Beispiel einer REST-Konformen Anwendung [Sam10]	11
Abb. 8	Beispiel einer State-Machine mit HATEOAS [Gie16]	13
Abb. 9	Message Channel	14
Abb. 10	Event-Driven Communication	15
Abb. 11	Stark vereinfachte Darstellung der Verwendung eines Service, der so- wohl Authentifizierung als auch Autorisierung anbietet.	17
Abb. 12	Auslastungsproblem bei der Skalierung von Monolithen	20
Abb. 13	Scaling der gleichen Komponenten in einer Microservice-Architektur .	21
Abb. 14	Architekturvorgabe it@M	24
Abb. 15	Kommunikation innerhalb der Architektur	25
Abb. 16	Aufbau eines Service innerhalb der it@M-Architektur	26
Abb. 17	Funktionale und nicht-funktionale Testmethoden	27
Abb. 18	Umfang der verschiedenen Testmethoden	28
Abb. 19	Unit Testing Scope [Cle14]	30
Abb. 20	Integration Testing Scope [Cle14]	31
Abb. 21	Component Testing Scope [Cle14]	32
Abb. 22	Contract Testing [Cle14]	34
Abb. 23	End-To-End Testing Scope mit externen Abhängigkeiten [Cle14] . . .	36
Abb. 24	Aufbau des Referenzsystems zur Testimplementierung	42
Abb. 25	Code coverage des Ordering-Service	44
Abb. 26	Code coverage des Shoppingcart-Service	44

IV Tabellenverzeichnis

V Listing-Verzeichnis

code/customer.json	12
../ReferenceSystem/.mdsd/referencesystem.barrakuda	40

VI Abkürzungsverzeichnis

HATEOAS	Hypermedia as the engine of application state
LDAP	Lightweight Directory Access Protocol
REST	Representational State Transfer
CORS	Cross-Origin Resource Sharing
CDCT	Consumer Driven Contract Testing
DDD	Domain-Driven Design
LHM	Landeshauptstadt München
RPC	Remote Procedure Call
API	Application Programming Interface
ORM	Object-Relational Mapping
IaC	Infrastructure as Code
GUI	Graphical User Interface
DNS	Domain Name System
URI	Uniform Resource Identifier
JPA	Java Persistence API
DSL	Domain Specific Language
VCS	Version Control System
IDE	Integrated Development Environment
CI	Continuous Integration
VM	Virtual Machine

1 Einführung und Motivation

Ziel dieser Arbeit ist es, mithilfe von generativer Testerstellung die Qualität einer Anwendung in Microservice-Architektur steigen zu lassen, ohne dabei die Kosten des Projekts zu erhöhen.

Sowohl im privaten als auch geschäftlichen Alltag nimmt IT 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 an die Qualität. Sobald Türklingeln, Alarmanlagen und Schließanlagen *smart* werden, ist die Fehlertoleranz gleich null. Auch Autos und medizinische Geräte sind nicht ausgenommen. In diesen Bereichen kann es sogar um Menschenleben gehen. Ganz abgesehen davon steigt auch die Komplexität von modernen Software-Systemen immens an, auch weil dies von den Nutzern der Anwendungen durch den Wunsch an neuen Features gefordert wird.[Pan99].

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(siehe Abb. 1).

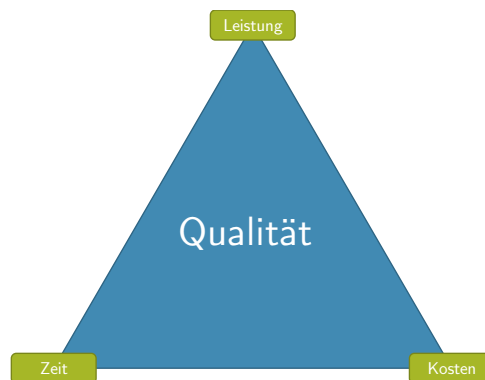


Abbildung 1: Magisches Dreieck des Projektmanagements [Hag08]

Dieses besagt, dass die Qualität eines Projekts durch die drei Faktoren Leistung, Kosten und Zeit beeinflusst wird. Diese Faktoren müssen vom Management eines Projekts möglichst ausbalanciert gehalten werden. Beispiel anhand des Hausbaus: Vom Bauherren ist ein fester Termin für die Fertigstellung des Hauses angedacht (Faktor Zeit), doch lässt der aktuelle Baufortschritt eine Fertigstellung zum festgelegten Termin nicht mehr zu. Die Lösung wäre, mehr Arbeiter einzustellen und somit den Fortschritt zu beschleunigen (Erhöhung der Kosten), oder die Arbeiten am Haus schneller und mit geringerer Qua-

litätsanforderung durchführen zu lassen (senkung der Qualität). Gleiches gilt auch für die Softwareentwicklung.

Zu beachten ist jedoch, dass eine hohe Qualität von Beginn des Projekts an angestrebt werden sollte, und nicht erst dann, wenn alle anderen Aufgaben erledigt sind. Ein Bericht der Kölner Beratungsfirma SQS zeigt anhand von gesammelten Zahlen aus Beratungsaufträgen welche immensen Kosten durch unentdeckte Fehler entstehen [SQS]. Hier wird besonders deutlich wie wichtig es für ein Projekt ist, frühzeitige Qualitätssicherung durchzusetzen. Und dazu zählt auch das Testen von Software.

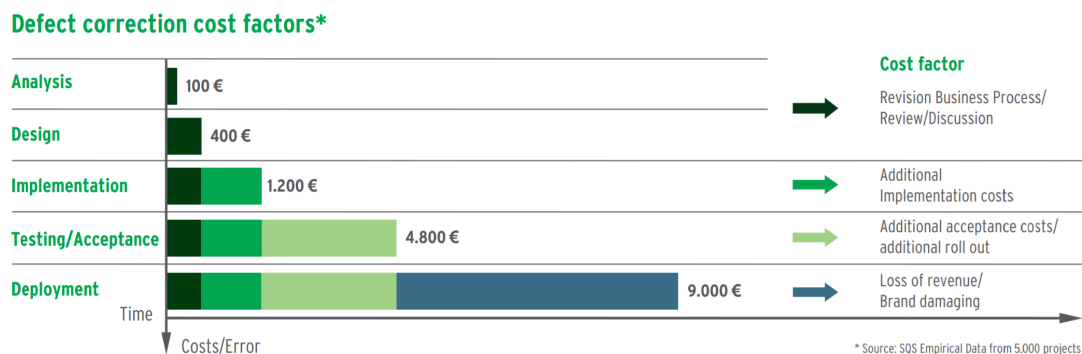


Abbildung 2: SQS Report Costs of Defect Correction [SQS]

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€ (siehe Abb. 2) 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. Hinzu kommt, dass gute Tests zur Steigerung der Qualität neben Geld auch Zeit kosten (siehe dazu Abb 1). Es entsteht in dieser Zeit aber kein Fortschritt an der Funktionalität der Software.

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 Barrakuda¹. 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 durch die Entwicklung von Testmethoden für Microservice-Architekturen der Grundstein für eine Erweiterung von Barrakuda gelegt 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 als Vorbereitung auf die darauf folgenden Kapitel die vorgegebene Architektur von it@M erläutert und auch die interne Struktur der Komponenten dargestellt (Kapitel 3).

Im nächsten Schritt werden bekannte Methoden zum Testen von Software festgestellt und analysiert (Kapitel 4.1). Dann werden, anhand der im vorherigen Kapitel festgestellten Eigenschaften von Microservice Architekturen, Test-Methoden, die besonders im Bereich dieses Architekturmodells sinnvoll zu verwenden sind, weiter ausgeführt und aufgestellt (Kapitel 5).

Abschließend werden diese Testmethoden im Kapitel der Implementierung (Kapitel 6) in einem Referenz-System, unter ständiger Beachtung der möglichen Generierbarkeit aller

¹Github Repository von Barrakuda (<https://github.com/xdoo/mdsd>)

Tests, implementiert und deren Effektivität und die mögliche Zeitersparnis analysiert. Am Ende wird im Fazit eine Beurteilung über die Sinnhaftigkeit und den möglichen Mehrwert zur Generierung der Tests aufgestellt (Kapitel 7).

2 Architekturvergleich Microservice und Monolith

Eine der momentan am meisten diskutierten Architekturen ist die „Microservice“ Architektur²: Viele kleine Services die im Konglomerat für ein gemeinsames Ziel zusammenarbeiten. Dabei gibt es insbesondere in der Kommunikation zwischen den Komponenten Unterschiede zwischen monolithischen und Microservice Architekturen. Während diese Komponenten in Monolithen gebündelt in einem Anwendungskontext agieren, passiert dies in Microservice-Systemen verteilt (vgl. Abb. 3).

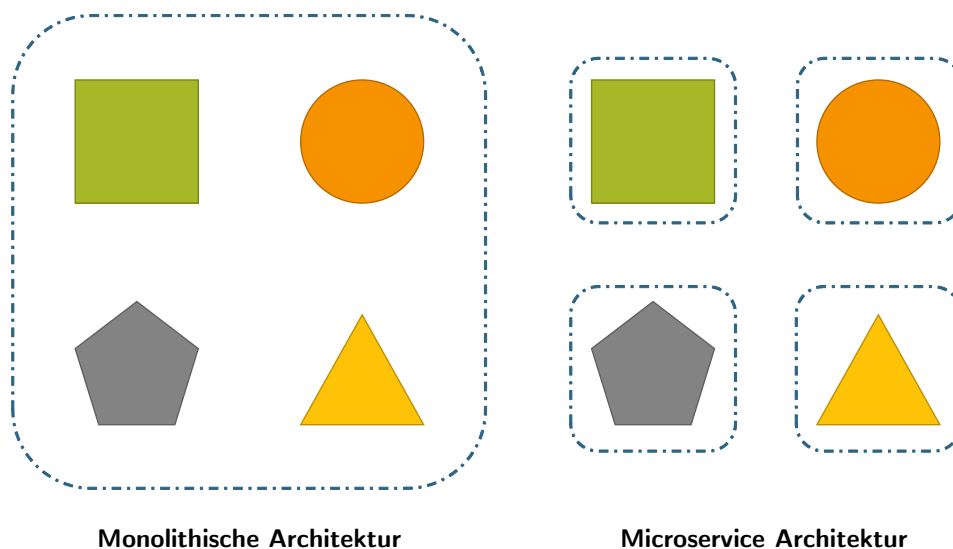


Abbildung 3: Logische Unterscheidung zwischen monolithischer und Microservice Architektur

Sam Newman stellt in seinem Buch *Building Microservices*[New15] 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<https://www.google.de/trends/explore?q=microservices> oder <https://jaxenter.de/ausgaben/java-magazin-5-16> oder <https://www.heise.de/developer/artikel/Microservices-im-Zusammenspiel-mit-Continuous-Delivery-Teil-1-die-Theorie-2376386.html?artikelseite=2>

2.1 Microservice Architektur

Microservices sind eine Art der Architektur, in der die Modularisierung im Vordergrund steht. Dazu werden komplexe Anwendungen in kleinere, einzelne Prozesse aufgeteilt die miteinander Sprachunabhängig in Verbindung stehen. [Wol15a, S. 2-3]

Um die Vorteile, die diese Architektur mit sich bringt, 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]

2.1.1 Definition von Service-Grenzen

Zum Planen einer Software mit Microservice Architektur ist es von großer Bedeutung, sich zunächst über die Grenzen seiner Services Gedanken zu machen. Als Ansatz zur Planung gibt es zwei bekannte Möglichkeiten.

2.1.1.1 Bounded Context Kontextgrenzen sind eine Definition aus dem Buch *Domain Driven Design* von Eric Evans [Eva03] und bieten eine Möglichkeit in der Planungsphase eines Projekts diese Grenzen zu definieren.

Dies lässt 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(siehe Abb. 4).

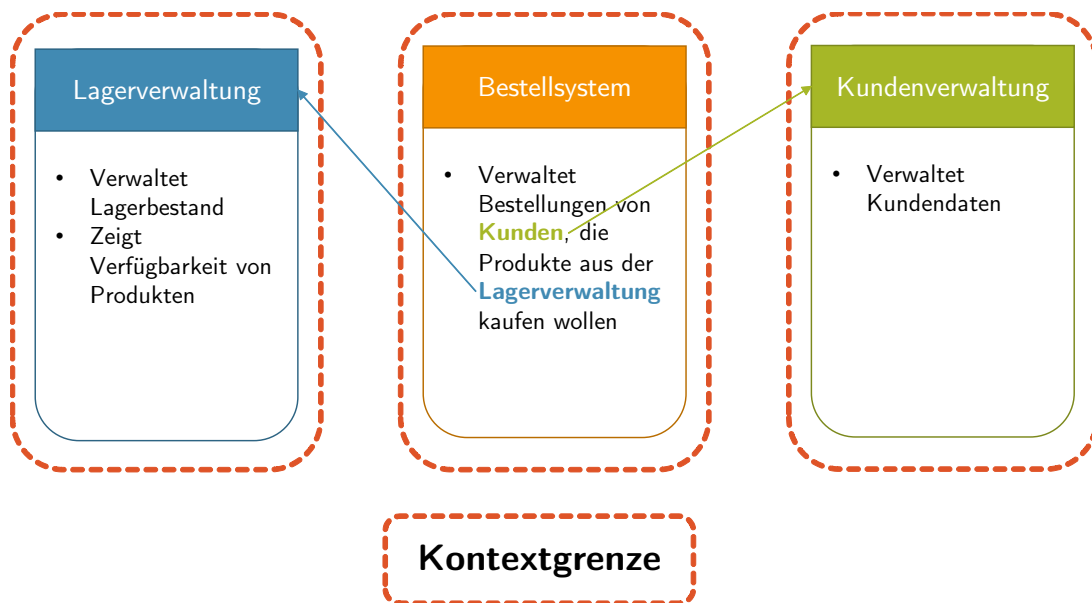


Abbildung 4: Beispiel der Kontextgrenzen des Online-Shops

Eine Kontextgrenze soll also eine logische Grenze darstellen, welche eventuell über eine 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]“

2.1.1.2 Aggregates Sind der zweite Mögliche Ansatz zur Planung von sinnvollen Service-Grenzen. Aggregates sind ein weiteres Konzept des Domain-Driven Design (DDD) und stellen eine Zusammensetzung aus Events und Objekten dar, die durch ein Aggregate Root zusammengehalten werden.

Als Beispiel dient hier ein Tisch der im Baumarkt gekauft wurde. Dieser Tisch besteht aus einer Sammlung von Holzstücken und Schrauben. Wenn diese auf einem Haufen liegen stellt dies noch lange keinen Tisch dar. Um den Tisch zu erhalten benötigt man eine An-

leitung (Events) die besagen, wie mit den Materialien umgegangen werden soll, um diesen Tisch zu erhalten. Diese Konstellation, von Materialien und der Anleitung zum Aufbau des Tisches, stellt ein Aggregate dar. Die Events alleine erfüllen keinen tieferen Zweck, auch die Materialien nicht. Aber vereint in einem Aggregate bilden sie den gewünschten Tisch.[Mog16]

Wenn man dieses Konzept nun auf das vorhergehende Online-Shop Beispiel überträgt, ergibt sich ein Bild wie in Abb. 5.

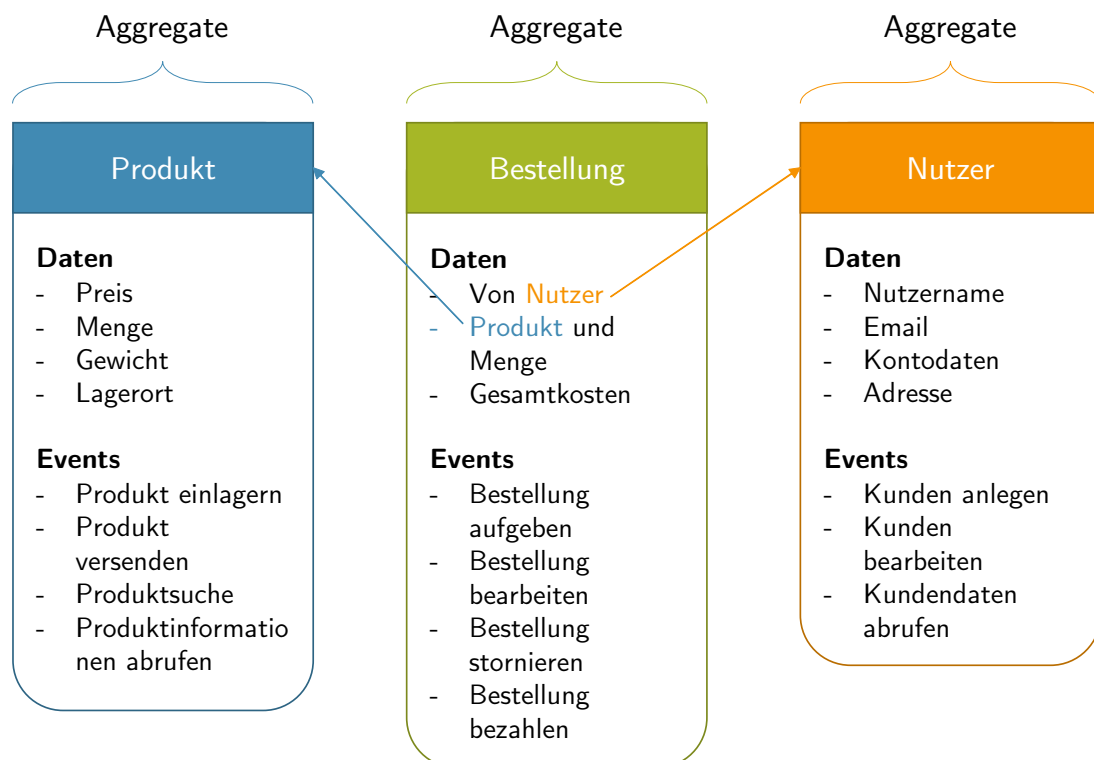


Abbildung 5: Aggregates im Online-Shop Beispiel

Wie zu sehen ist, decken sich die gefundenen Aggregates mit dem im vorherigen Abschnitt gefundenen Kontextgrenzen. Somit ist das Ergebnis der Aufteilung auch mit dieser Methode das gleiche. Aggregates machen insbesondere dann Sinn, wenn mit Event-basierter Kommunikation gearbeitet wird, da die Zustandsänderungen, die durch Events beschrieben werden, der Darstellung von Bearbeitungsregeln von Aggregates sehr nahe kommen.

2.1.2 Kommunikation zwischen Services

Als weiterer Schritt muss eine Kommunikationsart gewählt werden. Dabei gibt es drei mögliche Kommunikationswege für die verschiedene Technologien in Frage kommen:

- Services kommunizieren Untereinander

- Kommunikation mit einem Service von außen
- Service kommuniziert mit einem fremden System

Wobei der letzte Punkt an dieser Stelle vernachlässigt werden kann, da in diesem Fall das Fremdsystem die verwendete Technologie vorschreibt.

2.1.2.1 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].

2.1.2.2 Remote Procedure Calls (RPCs) sind eine weitere bekannte Kommunikationsmöglichkeit. Die Übersetzung ins deutsche erklärt schon einen Großteil der Idee hinter RPCs: „Aufruf 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 seinem RFC #707 „A High-Level Framework for Network-Based Resource Sharing“[Whi76]. Ein RPC funktioniert, indem der Client einer Anwendung einen Request 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].

Bei Verwendung von RPC werden in der Implementierung Stubs und Skeletons verwendet. Die Stubs sind dabei Client-seitige Schnittstellen, die es ermöglichen Prozeduren die auf einer anderen Maschine zur Verfügung stehen aufzurufen wie eine lokale Prozedur.

Der Stub sorgt dann für das marshalling (umwandeln) der Daten und den Transfer über das Netzwerk. Auf der Server-Seite empfängt das Skeleton, welches die gleichen Methodensignaturen wie der Stub bereithält, die Anfrage, sorgt für die Ausführung und liefert die Ergebnisse, ebenfalls über das Netzwerk, zurück (siehe Abb. 6). [Krz12]

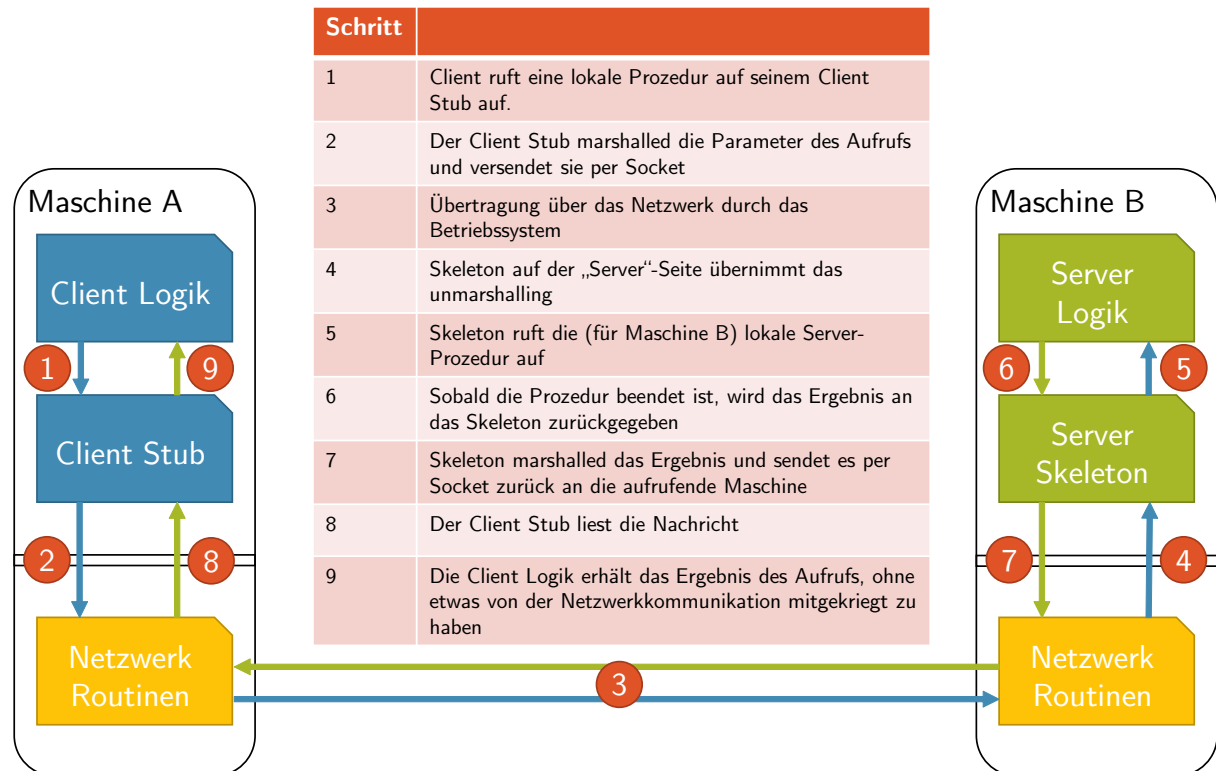


Abbildung 6: Ablauf eines RPC

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 Kundenobjekts: Zusätzlich zur Erstellung eines Kunden anhand 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.

2.1.2.3 Representational State Transfer (REST) ist eine der meistgenutzten Programmierparadigmen für Application Programming Interfaces (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 die folgenden 6 Eigenschaften, die erfüllt sein müssen um eine REST-Konforme Anwendung zu schreiben (ein Beispiel dazu in Abb. 7):

- Client-Server
- Stateless
- Cache
- Uniform Interface
- Layered System
- Code-On-Demand (optional)

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 Resource

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 nötig, 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 Informationen Hyperlinks zu anderen Informationen enthalten.

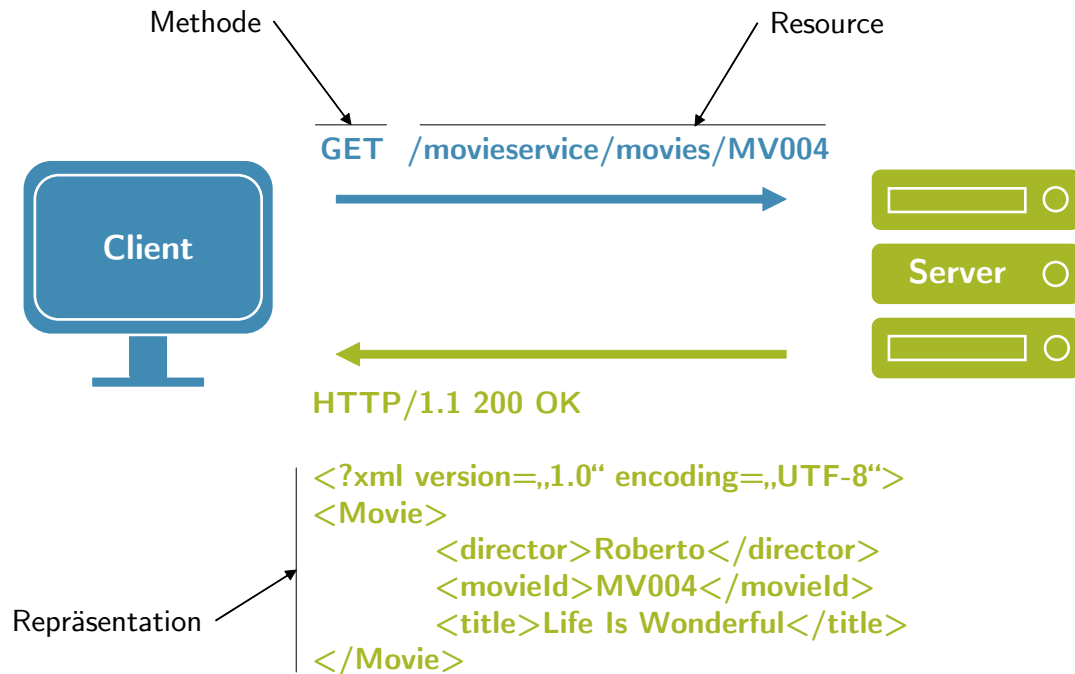


Abbildung 7: Beispiel einer REST-Konformen Anwendung [Sam10]

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-Protokolle 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.

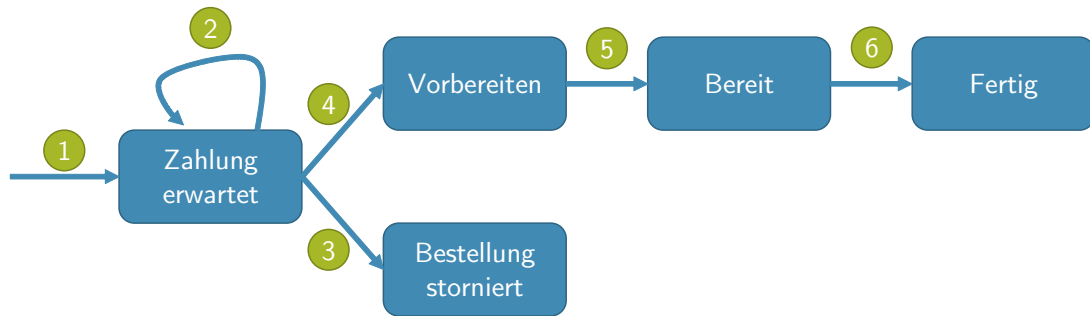
Die Verwendung von REST bietet viele Möglichkeiten die lose Kopplung zwischen Services

zu ermöglichen. Dazu zählt unter anderem HATEOAS. Um bei dem Kundenbeispiel aus 2.1.2.2 zu bleiben: Wird die Information eines Kunden abgerufen, kann das Kundenobjekt zusätzlich zu den Kundendaten ein Feld enthalten welches zur Bestellliste dieses Kunden zeigt.

```
1  {  
2    "name": "Hans Meier",  
3    "mail": "hansmeier@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 URIs erreichbar sind, oder sich interne Datenstrukturen ändern müssen diese Clients nicht neu angepasst werden.

Ein weiterer Anwendungsfall für HATEOAS ist die Nutzung der weiterführenden Links zur Bildung eines Zustandsautomaten (siehe Abb. 8). Die mitgelieferten Informationen zu weiteren *Ressourcen* helfen dabei dem Client einer Anwendung Manipulationsmöglichkeiten des empfangenen Objekts zu prüfen.



Schritt	Methode	URI	Aktion
1	POST	/orders	Bestellung erstellen
2	POST/PATCH	/orders/{id}	Bestellung bearbeiten
3	DELETE	/orders/{id}	Bestellung stornieren
4	PUT	/orders/{id}/payment	Bezahlen
5	GET	/orders/{id}	Bestellstatus zyklisch Abfragen
	GET	/orders/{id}/receipt	Quittung abrufen
6	DELETE	/orders/{id}/receipt	Bestellprozess abschließen

Abbildung 8: Beispiel einer State-Machine mit HATEOAS [Gie16]

Somit gibt die API bereits den späteren Prozess Endanwendung vor. Ein Nutzer könnte alleine durch dem einfachen folgen der mitgelieferten Links einen Ablauf entsprechend der Geschäftslogik abarbeiten.

2.1.2.4 Message Channel Neben den oben genannten Möglichkeiten der synchronen Kommunikation gibt es auch in der Welt der Microservices asynchrone Ansätze. Einer davon ist die der Messages. Anstelle des typischen Frage-Antwort-Ablaufs einer gängigen Client-Server-Kommunikation wird bei der Verwendung von Nachrichten auf Buffer gesetzt. Ein Service, der einen Auftrag an einen anderen Service senden möchte, schickt eine Nachricht an einen Message-Buffer. Der angesprochene Service bearbeitet die Nachrichten dann nach eigenem Gusto (siehe Abb. 9). Zu den Vorteilen dieser Art der Kommunikation zählt unter anderem, dass die Skalierung stark vereinfacht wird, da einfach weitere Konsumenten gestartet werden können, die Nachrichten aus dem Buffer abarbeiten [Wol15b]:

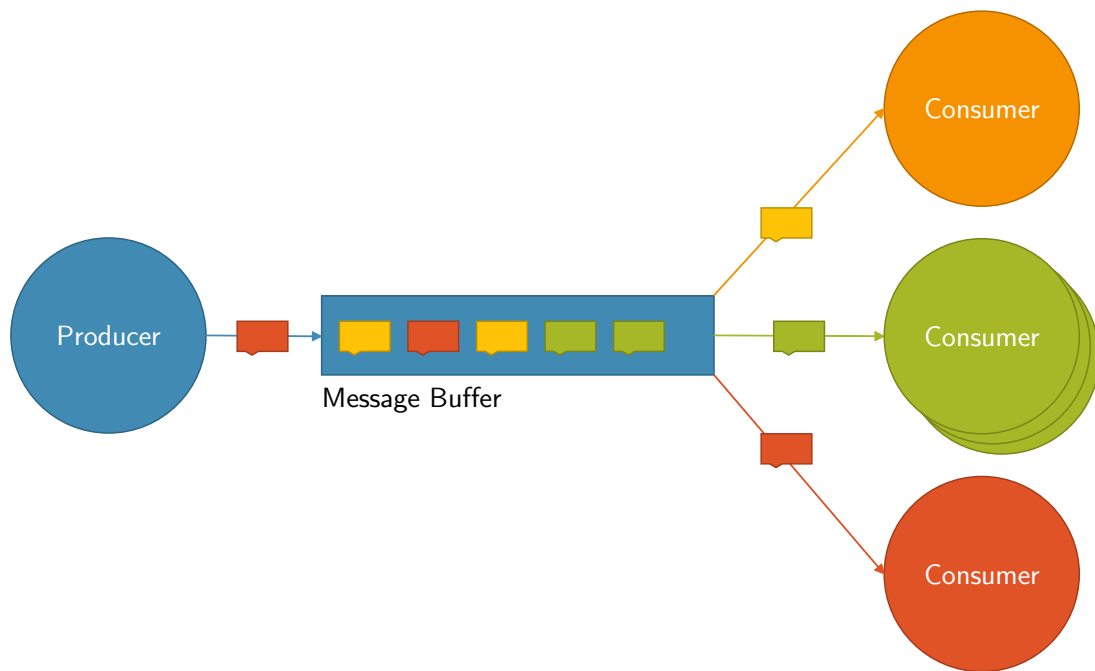


Abbildung 9: Message Channel

2.1.2.5 Event-Driven Communication ist die zweite Art der asynchronen Kommunikation. Der größte Unterschied zum Ansatz der Messages liegt darin, dass bei der Verwendung von Events kein expliziter Empfänger genannt wird, sondern alle Services ein Event empfangen, und eigenständig entscheiden, ob sie dieses Event verarbeiten wollen, oder nicht (siehe Abb. 10). Eines der größten Probleme dieses Ansatzes ist, dass der Empfang eines Events nicht sichergestellt werden kann, dafür ist es aber um einiges einfacher Use-Cases abzudecken, in denen Änderungen an einem Service an viele weitere propagiert werden müssen. [Wol15b]

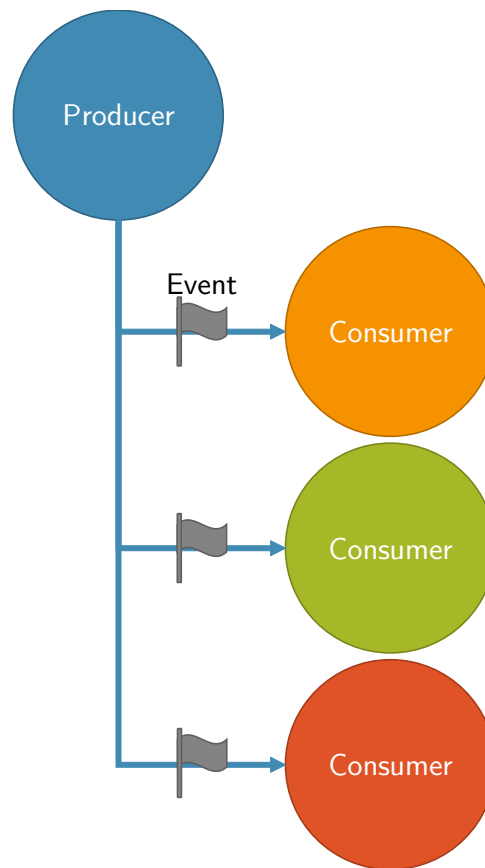


Abbildung 10: Event-Driven Communication

2.1.3 Datenformat - 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 Formate. 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) bietet. [New15, S.101].

2.1.4 Autorisierung mit OAuth2

Beim Thema der Clientautorisierung von Web-APIs ist OAuth2 weit verbreitet, insbesondere auf Social Media Plattformen. Ausschlaggebender Faktor dafür ist die Problematik, die sich bei verteilten Systemen mit Rechtesystemen gerne aufdrängt: Woher weiß ein Service, wer der anfragende Nutzer ist und was er darf? Dieses Problem soll möglichst so gelöst werden, dass es gut in die Philosophie der Microservices integriert werden kann. Eine eigene Autorisierung je existentem Service ist somit keine zufriedenstellende Lösung. OAuth2 bietet an dieser Stelle eine passende Lösung. Mit OAuth2 ist es möglich, die In-

formationen zu den Rechten eines Nutzers einfach über Services hinweg zu propagieren und die Forderung nach Zustandslosigkeit der Anfragen von REST (siehe Kapitel 2.1.2.3) nicht zu verletzen.

Wichtig ist hierbei die Unterscheidung zwischen Autorisierung und Authentifizierung. OAuth2 bietet keine Möglichkeit der Authentifizierung, also beispielsweise der Frage nach einem Nutzernamen und Passwort, sondern kümmert sich ausschließlich darum festzustellen, ob ein Nutzer die nötigen Rechte zum durchführen einer bestimmten Operation besitzt.[Deg15]

Allerdings ist es für die Verwendung von OAuth2 innerhalb eines Unternehmenskontext sinnvoll zusammen mit der Autorisierung auch eine Authentifizierungsmöglichkeit am Authentifizierungs-Service anzubieten. Dieser Service kann dann die bestehende Infrastruktur des Unternehmens zur Authentifizierung von Nutzern verwenden. Zusätzlicher Vorteil dieses Vorgehens ist, dass jeder Service der zusätzliche Informationen eines Nutzers erfragen möchte, dies an einer zentralen Stelle mithilfe des vom Nutzer propagierten Tokens einfach tun kann.

Ein Nutzer kann dann an einem Endpunkt des Authentifizierungsservice den OAuth2-Token erhalten, mit dem er sich an anderen Services autorisieren kann. Diese Services können einen weiteren Endpunkt nutzen, um etwa Nutzerinformationen anhand des Tokens abzurufen, die für eventuelle Identitätsanforderungen benötigt werden (siehe Abb. 11).

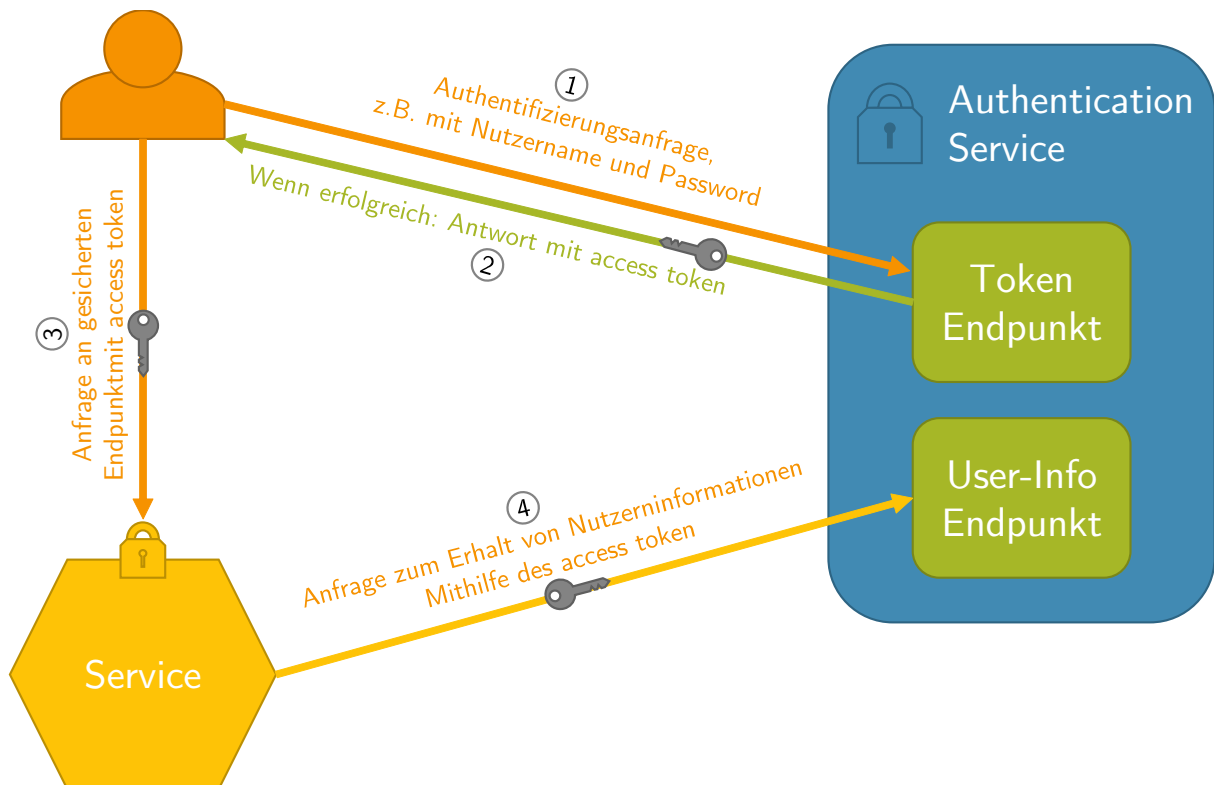


Abbildung 11: Stark vereinfachte Darstellung der Verwendung eines Service, der sowohl Authentifizierung als auch Autorisierung anbietet.

2.1.5 Deployment

Das Deployment ist in Microservice-Architekturen ein wichtiger Punkt, da eine gute Deployment-Infrastruktur auch für den Nutzen einer solchen Architektur entscheidend sein kann: keine Downtime, kurze Zeit bis Lösungen den Kunden erreichen und einfache Skalierung. Es gibt verschiedene Ansätze, wie man eine Anwendung zum Einsatz bringen kann[Ric16].

Multiple Service Instances per Host Bei diesem Ansatz werden ein, oder mehrere physikalische oder virtuelle Hosts genutzt, auf denen mehrere Instanzen von Services laufen.

Service Instance per Virtual Machine (VM) Jeder zu deployende Service wird als VM-Abbild gebündelt. Dieses Abbild wird dann zum Starten eines Services genutzt. Vorteil davon ist, dass jeder Service komplett isoliert von den anderen Services laufen kann. Ein Nachteil ist der hohe Overhead, der durch die Nutzung einer VM anfällt.

Service Instance per Container Die sogenannten Container sind ein Virtualisierungsansatz auf Betriebssystem-Ebene. Im Gegensatz zu regulären VMs sind Container aber

weitaus leichtgewichtiger und erzeugen einen weitaus geringeren Overhead.

2.2 Monolithische Architektur

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. Dies wird durch verschiedene Module erreicht die nur über definierte Schichten hinweg miteinander kommunizieren. In der Entwicklung werden diese Module auch gerne physikalisch durch Auslagerung in Bibliotheken getrennt, um ein Abweichen von den vorgeschriebenen Kommunikationswegen zu unterbinden.

Entwicklungstools und IDEs unterstützen den Entwickler beim programmieren von monolithischen Systemen - historisch bedingt - sehr gut. Das Deployment ist sehr einfach. Beispielsweise reicht es bereits, ein einfaches WAR-File auf einem Application-Server auszuliefern. Auch die Skalierung ist einfach möglich, durch die Verwendung eines Load-Balancers und das starten von mehreren Instanzen der Anwendung.[Ric14]

2.3 Vorteile von Microservices gegenüber Monolithen

Als Vorteile der Microservice-Architektur lassen sich nun folgende Punkte, die unter anderem auch von Newman in seinem Buch dargestellt werden, festhalten[New15]:

Klare Team-Organisation Eine Gruppe von Entwicklern ist für die Entwicklung und Pflege eines jeden Service verantwortlich, anstatt ein großes Team zu einzelnen Teilkomponenten einer monolithischen Anwendung zuzuweisen.

Heterogenität welche es erlaubt, für verschiedene Einsatzzwecke verschiedene Sprachen, Technologien sowie Sprach- und Technologieversionen zu verwenden, ohne das ganze System damit implementieren zu müssen. Dieser Punkt ist insbesondere Interessant, wenn Firmen zusammengeführt werden, deren Anwendungen auf verschiedenen Sprachen basieren.

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 führen kann, weshalb in solchen Architekturen auch häufig kaum Änderungen durchgeführt werden. Dazu zählt auch die Eigenschaft, dass die Aktualisierung von Frameworks wesentlich einfacher von statten gehen kann. Während dies bei einem monolithisches System fast ein eigenständiges Projekt mit mehreren Tagen Arbeit verursacht, ist es möglich solche Updates bei Microservice-

Architekturen Schrittweise in kleineren Aufgabenpaketen zu erledigen, die auch zwischen größeren Aufgaben bearbeitet werden können.

Schnelleres Deployment Insbesondere kleine 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. Ein weiterer Vorteil vom schnelleren Deployment ist die Möglichkeit, durch Continuous Delivery die verlangte Code Coverage von Tests geringer zu halten. Dies ist möglich, da Fehler im Produktivsystem schnell behoben werden können, und keine Tagelangen Ausfälle durch einen komplizierten und Langwierigen Bereitstellungsprozess entstehen.

Erhöhte Widerstandsfähigkeit gegenüber Fehlern im System Die harten Grenzen von Microservices können eine Kaskadierung von Fehlern verhindern und somit entsteht bei korrekter angewendeter Architektur kein Gesamtausfall des Systems. Diese wird allerdings auch benötigt, wie in 2.4 nachzulesen ist.

Bessere Skalierung Auch wenn die Skalierung von Monolithen 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.

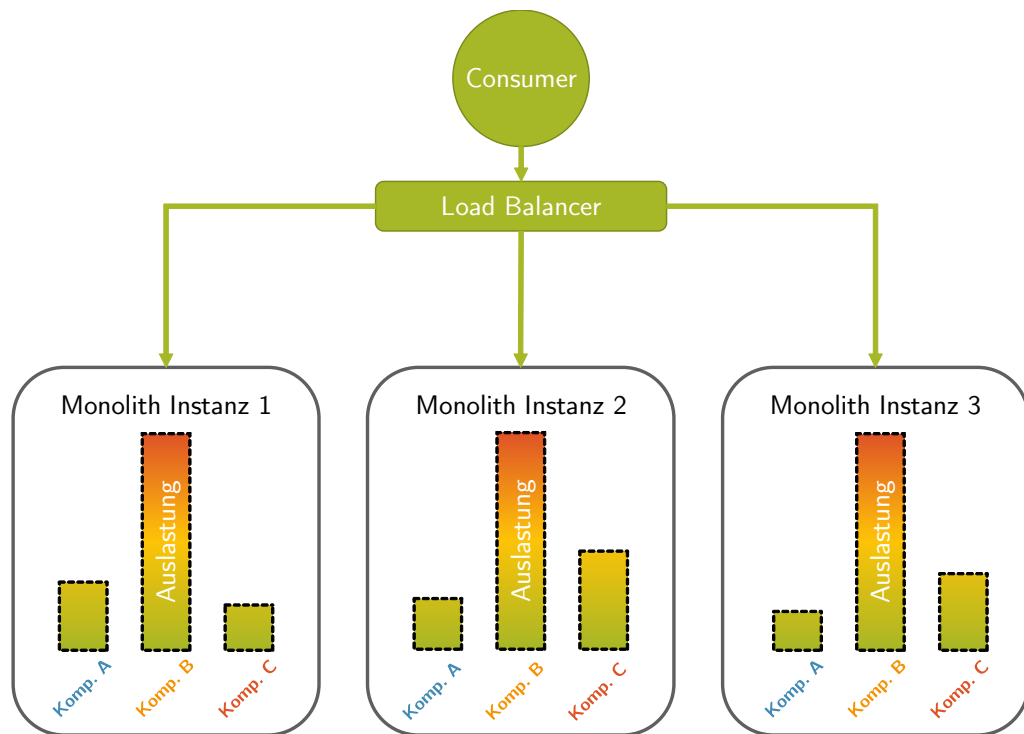


Abbildung 12: Auslastungsproblem bei der Skalierung von Monolithen

Microservices lassen sich dagegen effizienter skalieren. Während monolithische Architekturen immer im ganzen skaliert werden, kann man in einer Microservice-Architektur genau die Services skalieren, die in diesem Moment mehr Leistung benötigen (vgl. Abb. 12 und Abb. 13).

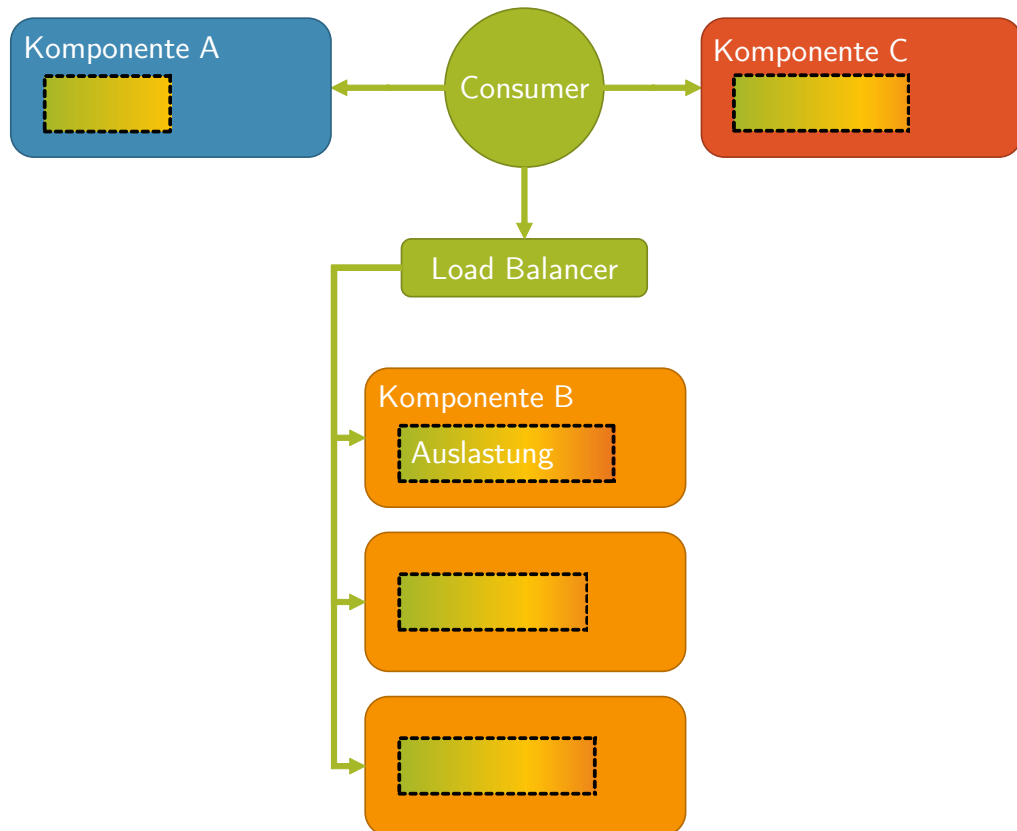


Abbildung 13: Scaling der gleichen Komponenten in einer Microservice-Architektur

Zusätzliche Schwierigkeiten von Monolithen Weitere Probleme nehmen bei Monolithen ab einem gewissen Entwicklungsstand 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.

Dies sind einige der Gründe, warum über Microservices so viel diskutiert wird und weshalb sich diese Arbeit insbesondere auf die Testerstellung von Anwendungen in dieser Architektur konzentriert. Besonders die genannten Vorteile der Skalierung, die Austauschbarkeit von verschiedenen Sprach- und Technologieversionen und die erhöhte Fehlertoleranz sind Hauptbeweggründe, warum auch 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

Doch gibt es, wie bei jeder Architektur, nicht nur Vorteile.

Verteilung Verteilte Systeme sind durch die Art der Kommunikation, also über das Netzwerk, einer weiteren Fehlerquelle ausgesetzt und vor allem eins: langsamer als Kommunikation innerhalb eines geschlossenen Systems[Fow15].

Hohe Fehlertoleranz benötigt Die Komplexität der laufenden Services auf der Infrastruktur ist ein weiterer negativer Aspekt, den man jedoch auch positiv auffassen kann. Es wird durch die Modularität und lose Kopplung der Komponenten eine hohe Fehlertoleranz von den einzelnen Teilen des Systems abverlangt. Es muss mit viel mehr Fehlerquellen umgegangen werden, als es bei einem Monolithen der Fall ist. So zum Beispiel der Ausfall oder die nicht-Erreichbarkeit von anderen Services. Dies führt allerdings auch dazu, dass viel Zeit in die Ausfallsicherheit der einzelnen Komponenten gesteckt werden muss, was eine höhere Qualität zur Folge hat. Netflix nutzt, um diese Ausfallsicherheit sicherzustellen und zu testen, ein eigens Entwickeltes Tool namens „Chaos Monkey“³. Dieses Werkzeug sorgt für zufällige Ausfälle von Komponenten in der produktiven Infrastruktur.

Eventual Consistency Dabei handelt es sich um ein weiteres Problem von verteilten Systemen. Daten über mehrere Instanzen eines Services Konsistent zu halten ist nahezu unmöglich. Es gibt keine hundert prozentige Sicherheit, dass alle Daten einer Anwendung überall aktuell verfügbar sind.

2.5 Resümee des Vergleichs

An dieser Stelle sollte noch erwähnt werden, dass die Idee hinter Microservices nichts neues ist. Bereits auf der Schicht des Betriebssystems wird dies deutlich. Auf UNIX-Systemen laufen lauter keine Prozesse und Dienste, die Unabhängig voneinander existieren und entwickelt wurden, und in ihrer Gesamtheit das Betriebssystem bilden.[Hof14]

Auch sind Monolithische Architekturen nicht gleich schlechter oder unnutzbar. Bestes Beispiel einer aktuellen *Start-Up* Firma ist etsy⁴, die ihren Online-Flohmarkt in Form einer Monolithischen Anwendung entwickeln und erfolgreich betreiben. Nicht zu vergessen sind

³<https://github.com/Netflix/SimianArmy>

⁴<https://www.etsy.com/about/?ref=ftr>

auch etablierte Softwaresysteme, die *trotz* der monolithischen Architektur sehr erfolgreich sind. Dazu zählt als bekanntestes Beispiel SAP⁵.

Ebenfalls zu ergänzen ist, dass, auch wenn Microservices zunächst klein und kompakt wirken, auch diese eine hohe Komplexität beinhalten können. Nur ist diese Komplexität versteckt vor den Entwicklern die nicht daran arbeiten und es handelt sich meist um fachliche Komplexität, während Monolithen meist eine strukturelle Komplexität vorliegt. Das aufteilen von Logik und Kernkompetenzen ist auch auf reiner Code-Basis möglich und wird auch so umgesetzt, solange sich an Interface-Definitionen und eine Strenge Trennung von verschiedenen Komponenten eines Monolithen gehalten wird.[Hof14]

3 Vorgegebene Architektur der Landeshauptstadt München (LHM)

3.1 Backing Services der Architektur

Von it@M sind bereits Entscheidungen zum Aussehen einer Anwendung in Microservice-Architektur getroffen worden. Als Infrastrukturelle Unterstützung gibt es innerhalb der Anwendung einen Authentifizierungs-Service, der sowohl eine Authentifizierung über die städtische Infrastruktur bietet, als auch über eine eigene Nutzerverwaltung (siehe Abb. 14).

Des weiteren existiert ein Discovery-Service, der im System wie eine Art DNS agiert⁶. Dieser Service erlaubt es allen im System beteiligten Services eine lokales Abbild seines Registers aller verfügbaren Services abzurufen, und über dieses Register Services anzusprechen. Im Gegensatz zu einem normalen Domain Name System (DNS) muss der Discovery-Service allerdings nicht manuell mit Daten gespeist werden, sondern verlangt es, dass Services sich eigenständig dort registrieren. Sobald also ein neuer Service innerhalb des Systems startet, meldet dieser sich mit seinem Namen und Netzwerkadresse an und ist ab sofort für die anderen Services verfügbar.

Zusätzlich existiert ein Configuration Service der die zentrale Steuerung von gemeinsamen Einstellungen erlaubt. Die größte Varianz bei Vergleich von Entwicklungs-, Test- und Produktionsumgebungen existiert in den Konfigurationen der Services. Dazu zählen u.a. Anbindungen an Datenbanken oder auch die IP-Adressen der zuvor genannten Backing Services[Wig12]. Ein Configuration Service lagert diese Konfigurationen aus dem Source

⁵<http://www.sap.com/germany/index.html>

⁶Wird für die Orchestrierung der Services in Produktion Kubernetes (<https://kubernetes.io/>) verwendet, wird tatsächlich ein DNS-Service zur Service-Discovery genutzt. Jede Instanz eines Service erhält einen DNS-Namen und die Adresse ist darüber auflösbar.(siehe dazu <https://kubernetes.io/docs/admin/dns/>)

Code der Services selber aus, um Konfigurationsänderungen an diesen Parametern auch zur Laufzeit des Systems ohne ein neues Deployment zu ermöglichen.

Des weiteren gibt es einen Edge Server. Dieser fungiert als eine Art Gateway. Er bietet eine Schnittstelle für die Gesamtanwendung und verhindert somit, dass die Komplexität der vielen kleinen Services für weitere Anwendungen und Nutzer nach außen getragen wird. Dies ist auch insbesondere dann wichtig, wenn das Frontend über Javascript mit den Services des Systems kommunizieren soll, da somit keine Cross-Origin Resource Sharing (CORS)-Anfragen mehr benötigt werden, weil auch das Frontend über diesen Edge Server bereitgestellt werden kann.

Neben zur Grundstruktur kann es in einer Anwendung mehrere Services geben, die alle innerhalb ihrer Kontextgrenzen Aufgaben erfüllen.

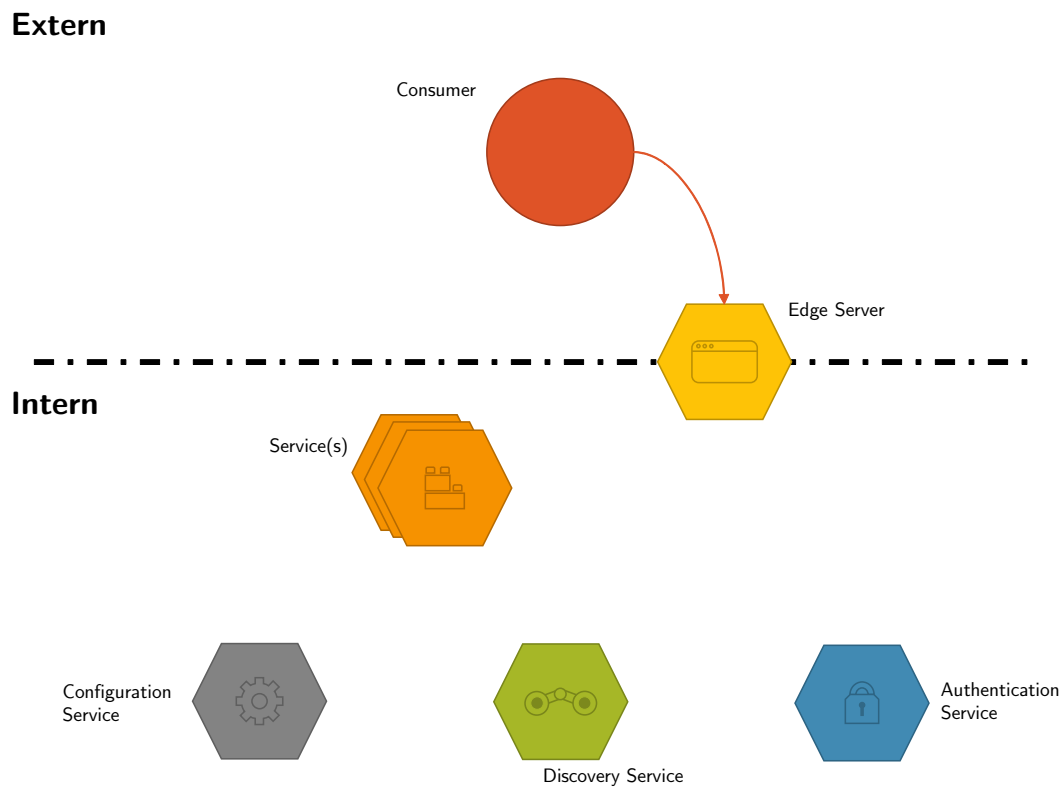


Abbildung 14: Architekturvorgabe it@M

3.2 Kommunikation zwischen den Komponenten

Zum Start der Anwendung registrieren sich alle Komponenten beim Discovery Service um für Anfragen erreichbar zu sein.

Der Konsument der Anwendung, sei es ein Nutzer oder eine weitere Software, kommu-

niziert ausschließlich über den Edge Server mit den internen Komponenten. Zunächst besteht für den Konsumenten die Möglichkeit, sich beim Authentication Service zu Authentifizieren und mithilfe eines OAuth2 Tokens für nachfolgende Anfragen zu Autorisieren. Bei Eintreffen eines Requests wird über den Discovery Service der zur Bearbeitung der Anfrage benötigte Service gesucht und seine Uniform Resource Identifier (URI) aufgelöst. Die Abfrage wird dann an den entsprechenden Service weitergeleitet. Bei allen gesicherten Endpunkten innerhalb der Anwendung findet immer eine Absprache mit dem Authentifizierungsservice statt, um auf eine Änderung der Rechte eines Nutzers sofort reagieren zu können. Bei ungesicherten Ressourcen ist diese Abstimmung natürlich nicht nötig (siehe Abb. 15).

Der Configuration Service aktualisiert bei Bedarf die Konfigurationen aller beteiligten Services innerhalb der Anwendung.

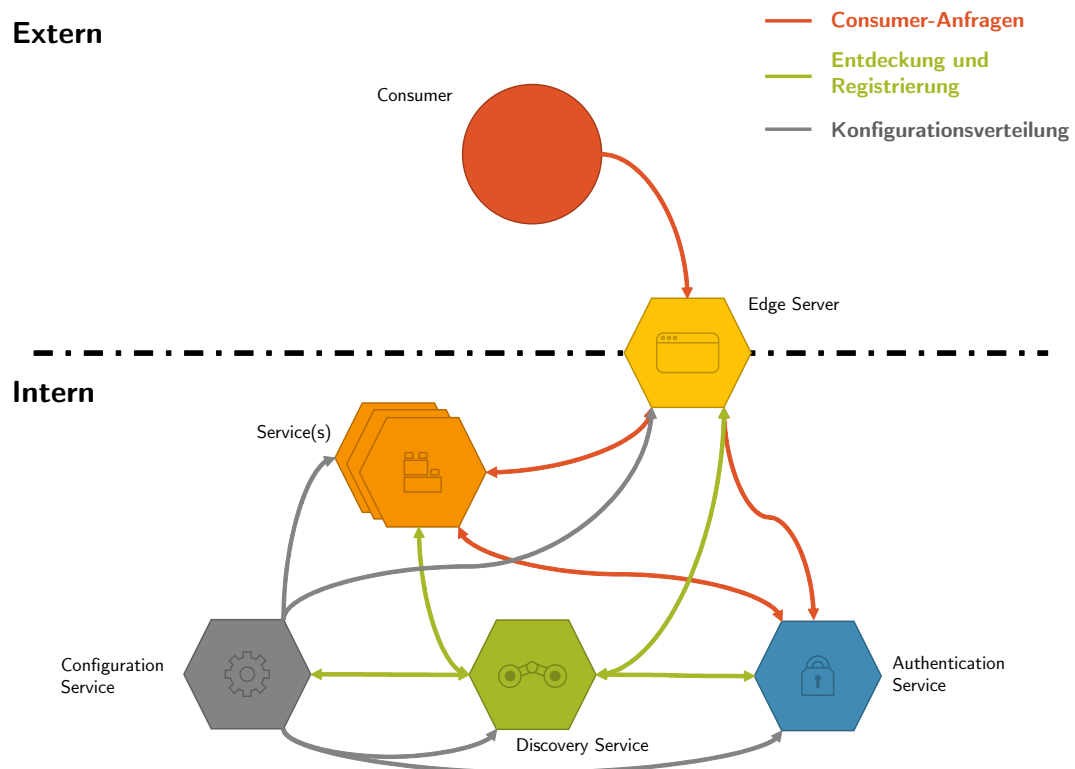


Abbildung 15: Kommunikation innerhalb der Architektur

3.3 Interne Struktur der Services

Jeder Service der Anwendung hat intern die gleiche Aufteilung. In Abbildung 16 ist eine graphische Darstellung zu finden. Ziel ist es, für eine Standardisierung innerhalb der Services zu sorgen, um Entwicklern den Einstieg zu erleichtern, egal an welchem Projekt aktuell Microservices entwickelt werden. Nach außen werden, REST-konform, nur

die Ressourcen über URIs angeboten. Diese Ressourcen bestehen entweder aus Entitäten oder Geschäftsanwendungen. Entitäten werden über sogenannte Repositories als Ressource bereitgestellt und über eine JPA-Bibliothek in einer Datenbank persistiert. Der Begriff des Repositories stammt ebenfalls aus der Welt des DDD und lässt sich anhand einer Definition von Edward Hieatt and Rob Mee gut erläutern[HM03]:

„Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.“

Ein Repository stellt also eine Verhandlungsebene zwischen den Schichten der Domäne und der Datenpersistierung. Zum zusätzlichen Eingriff durch Entwickler kann auf verschiedene Events mithilfe von Listeners reagiert werden. Zur Kommunikation mit anderen Services existiert ein *Security REST Client*, der diese für den Entwickler, durch automatische Verwendung des Discovery und Authentication Service, so einfach wie möglich machen soll.

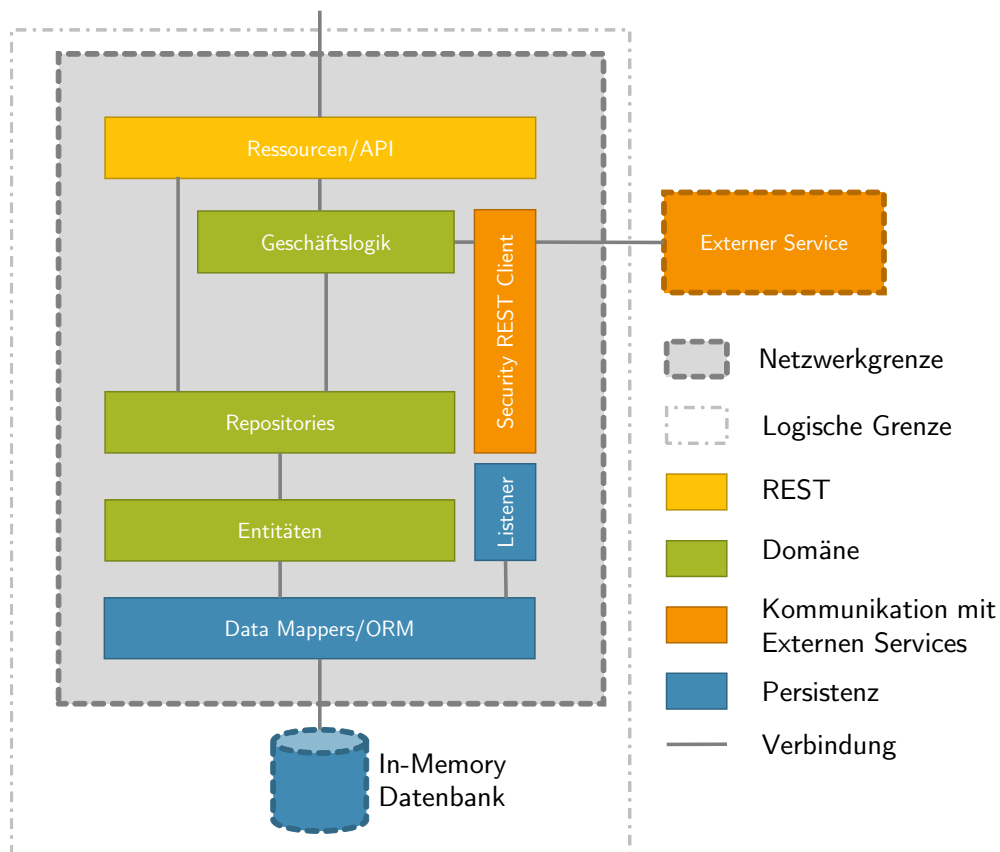


Abbildung 16: Aufbau eines Service innerhalb der it@M-Architektur

4 Software Testen

Das Testen von Software ist der Prozess, ein Programm oder einen Programnteil auszuführen mit dem Ziel Fehler zu finden und zu beheben. Im Gegensatz zu physikalischen Systemen kann Software auf unterschiedliche und unvorhersehbare Weisen defekte aufweisen. Dies liegt unter anderem daran, dass die vielen Fehler in Anwendungen nicht durch Fehler in Produktionsabläufen, sondern durch falsche oder schlechte Design-Entscheidungen entstehen.[Pan99]

4.1 Bekannte Testmethoden

Zunächst kann das Testen von Software in Funktionale und nicht-funktionale Tests getrennt werden (siehe Abb. 17). Zur weiteren Erklärung müssen zunächst die Begriffe der funktionalen und nicht-funktionalen Anforderungen beschrieben werden. Einfach gesagt beschreiben funktionale Anforderungen *was* das System tun sollte (Kundenanforderungen) und nicht-funktionale Anforderungen *wie* das System funktioniert (Technische Anforderungen).[Eri12a]

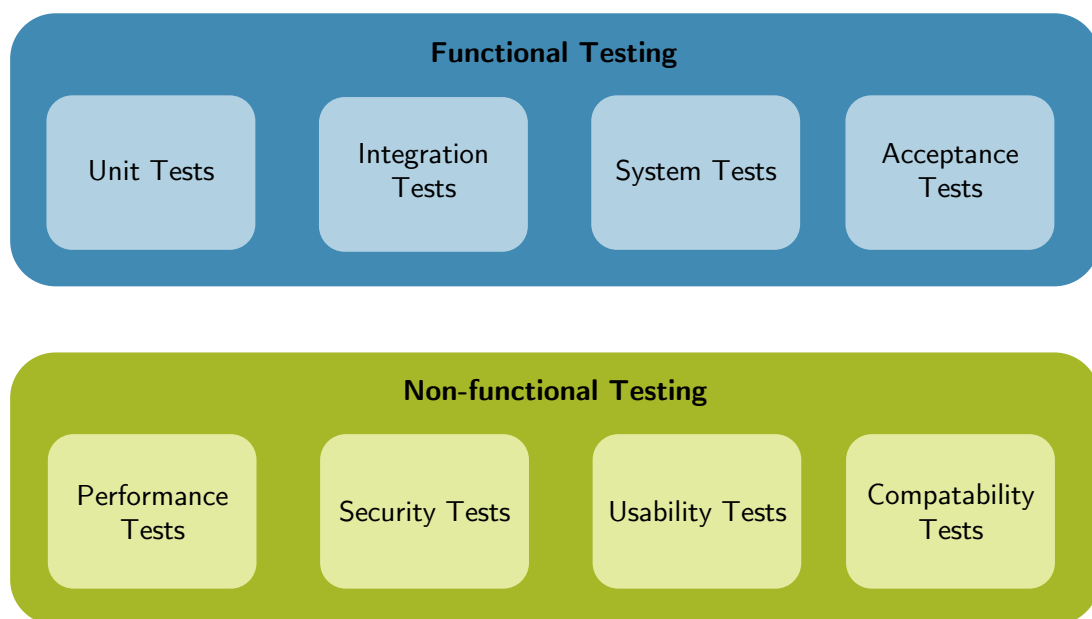


Abbildung 17: Funktionale und nicht-funktionale Testmethoden

Funktionale Tests beschäftigen sich nur mit den funktionalen Anforderungen einer Anwendung und stellen fest, ob, und wie gut diese Anforderungen erfüllt werden.[Eri12b]. Funktionales Testen wird häufig in vier Komponenten aufgeteilt, die auch meist in Reihenfolge abgearbeitet werden (siehe Abb. 18).[Inf16]

Unit Tests testen einzelne Module einer Software und werden meist direkt vom Entwick-

ler dieses Teilmoduls implementiert. Sie bewegen sich meist auf Klassenebene um die Komponenten so klein wie möglich zu halten. In testgeleiteten Programmierparadigmen werden diese Tests bereits vor der Implementierung der eigentlichen Komponente umgesetzt.[Inf16]

Integration Tests sind der nächste Schritt, wenn mehrere Komponenten durch Unit Tests abgedeckt sind. Diese getesteten Teile der Anwendung werden dann zusammengeführt und ihre Zusammenarbeit wird geprüft.[Inf16]

System Tests testen abschließend das Gesamtsystem. Also das Zusammenspiel aller Unit- und Integrationsgetesteten Teilkomponenten.[Inf16]

Acceptance Tests sollen sicherstellen, dass alle Produkt und Projekt-Anforderungen zur Zufriedenheit des Kunden erfüllt wurden. Sie stellt die finale Phase der funktionalen Tests dar.[Inf16]

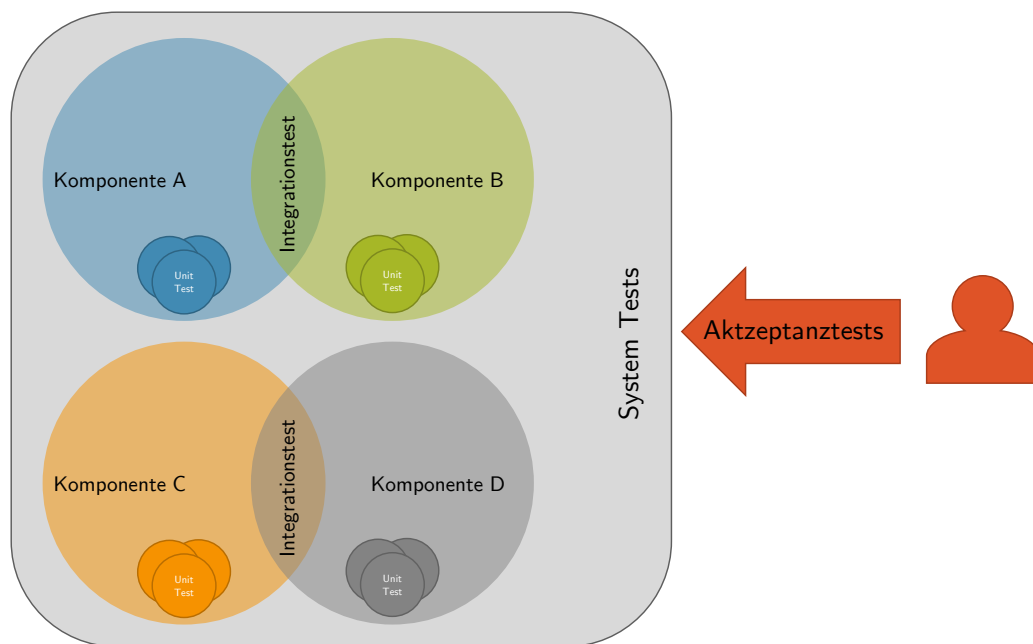


Abbildung 18: Umfang der verschiedenen Testmethoden

Nicht-funktionales Testen evaluiert die *Bereitschaft* des Systems[Eri12b]. Zu den nicht-funktionalen Testmethoden zählen unter anderem die folgenden[Inf16]:

Performance-Tests Mit diesen wird festgestellt, wie gut eine Anwendung mit vielen An-

fragen umgehen kann.

Security Testing konzentriert sich auf das Prüfen der Anwendung in Hinblick auf Vertraulichkeit von Informationen, Integrität, Erreichbarkeit und die Authentifizierung.

Usability Tests sind eine eher objektive Testmethode mit deren Hilfe die Benutzerfreundlichkeit einer Anwendung sichergestellt werden soll. Es wird besonders auf die Erlernbarkeit der Anwendung, Effizienz bei der Nutzung, Zufriedenstellung des Nutzers und Einprägsamkeit Wert gelegt.

Compatibility Tests Tests, die prüfen, ob die Anwendung auf allen geforderten Betriebssystemen, Browsern und/oder Hardware-Plattformen lauffähig ist und alle Funktionen sich kongruent verhalten.

5 Konzeption: Testen von Microservices und generative Testerstellung

Durch einige spezielle Eigenschaften von Microservices ergeben sich teilweise auch spezielle Testanforderungen die in der Entwicklung von Tests Beachtung finden müssen. Allerdings wird sich auch bei Microservices auf altbekannte Methodiken gestützt, insbesondere in den grundlegenden Bereichen der Unit- und Integrationstests.

5.1 Unit Testing

Auch bei Microservices beginnt das Testen eines Gesamtsystems auf der niedrigsten Schicht. Ebenso gilt es, dass Unit Tests generell auf Klassen- oder Methoden-Level geschrieben werden[Cle14]. Im Falle des Architekturmodells von it@M beschränken sich Unit Tests auf das Testen der Geschäftslogik, sowie der verschiedenen Listenern (siehe Abb. 19). Die Repositories und Entitäten sind durch das verwendete Framework nicht mit reinen Unit Tests testbar, sondern werden erst in den folgenden Integrationstests getestet.

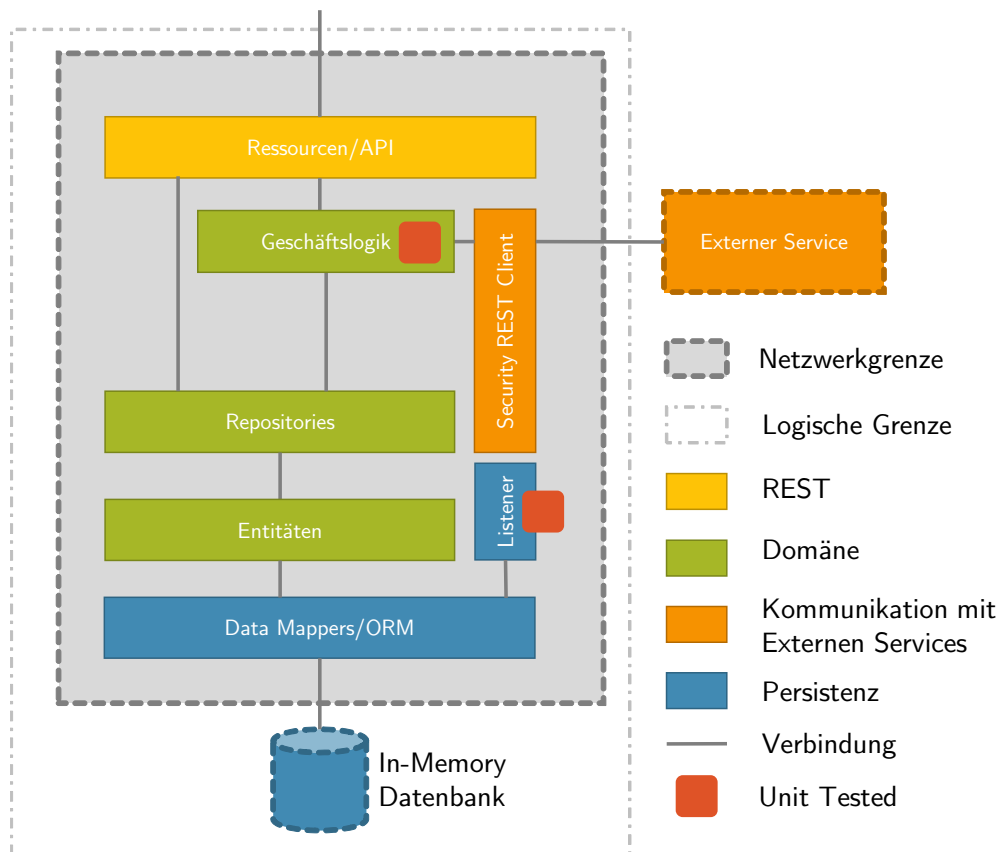


Abbildung 19: Unit Testing Scope [Cle14]

5.2 Integration Testing

An dieser Stelle ist die Kernfunktionalitäten der Services, also die Stellen, an denen nicht reiner Code zur Persistierung und Bereitstellung von Schnittstellen, sichergestellt. Doch die korrekte Zusammenarbeit der getesteten Komponenten ist noch nicht gewährleistet. Bei den Integrationstests werden nun mehrere Module zusammengefasst um das Zusammenspiel dieser Komponenten zu testen.[Cle14] Auch hier wird also nicht vom Regelfall aus Kapitel 4.1 abgewichen. In diesem Stadium wird das Zusammenspiel des Data Mappers und der angebundenen Datenbank, sowie die daraus resultierenden Ressourcen auf Korrektheit geprüft (siehe Abb. 20). Wichtig an dieser Stelle ist es, aufgrund der fortschrittlichen Implementierung von modernen Object-Relational Mappings (ORMs), sicherzustellen, dass die Testdaten nicht nur temporär vorgehalten wurden, sondern tatsächlich in die Datenbank geschrieben und von dort herausgelesen wurden, um etwaige Fehler auf diesen Schichten auszuschließen.[Cle14]

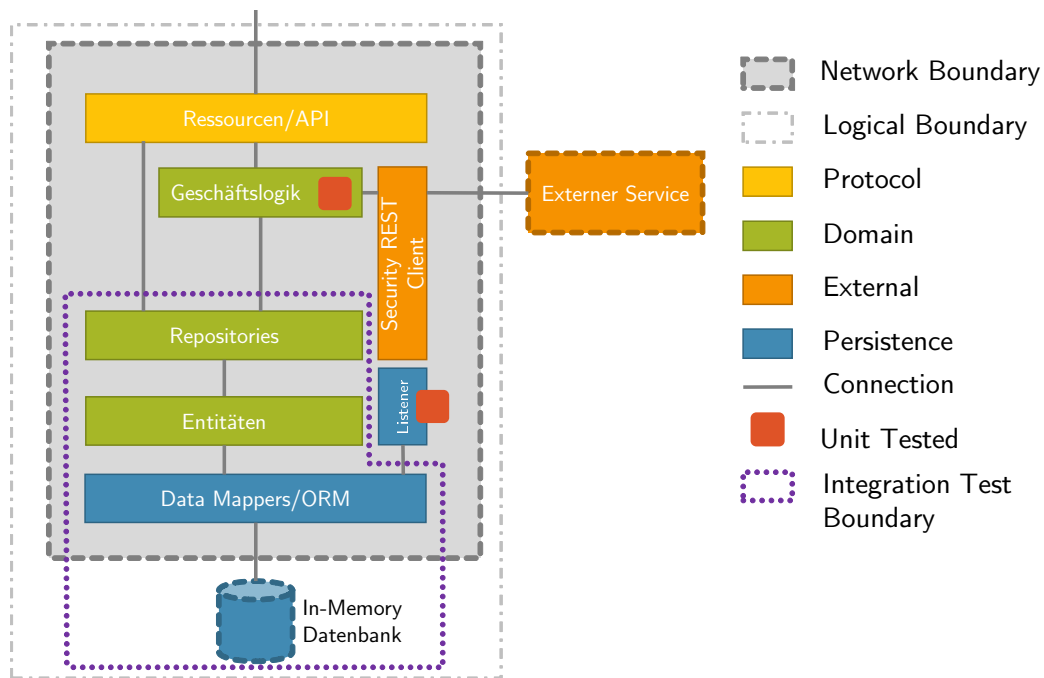


Abbildung 20: Integration Testing Scope [Cle14]

5.3 Component Testing

Unit- und Integrationstests sorgen dafür, dass an dieser Stelle bereits sichergestellt ist, dass die Logik in den einzelnen Komponenten garantiert so funktioniert wie sie beabsichtigt ist. Doch ist dies der Punkt, an dem sich das Testen von Microservices vom Testen von monolithischen Anwendungen in Details unterscheidet. Da der nächste Schritt nach dem Testen der einzelnen Teilkomponenten eines Service darin besteht, den gesamten Service zu testen, treten Probleme auf, sobald dieser zur korrekten Funktionalität externe Abhängigkeiten besitzt.

Für diesen Teil gibt es das Component Testing. Es eignet sich besonders gut für Microservice Architekturen, da diese sich leicht in die getrennt zu testenden Komponenten aufteilen lassen: Jeder Service entspricht einer Komponente. Es werden die Schnittstellen des einzelnen Service angesprochen und seine Funktionalität geprüft. Externe Kommunikationspartner werden durch *Test Doubles* ersetzt, dadurch wird eine Isolierung des Service für den Test sichergestellt (siehe Abb. 21).[Cle14] Ein *Test Double* ist eine generische Umschreibung für ein Objekt welches man zu Testzwecken gegen ein produktive Komponente austauscht. Dazu zählen u.a. *Dummys*, die zwar weitergereicht, aber nie verwendet werden, oder auch *Stubs*, die für bestimmte vordefinierte Anfragen vordefinierte Antworten liefern.[Fow06]

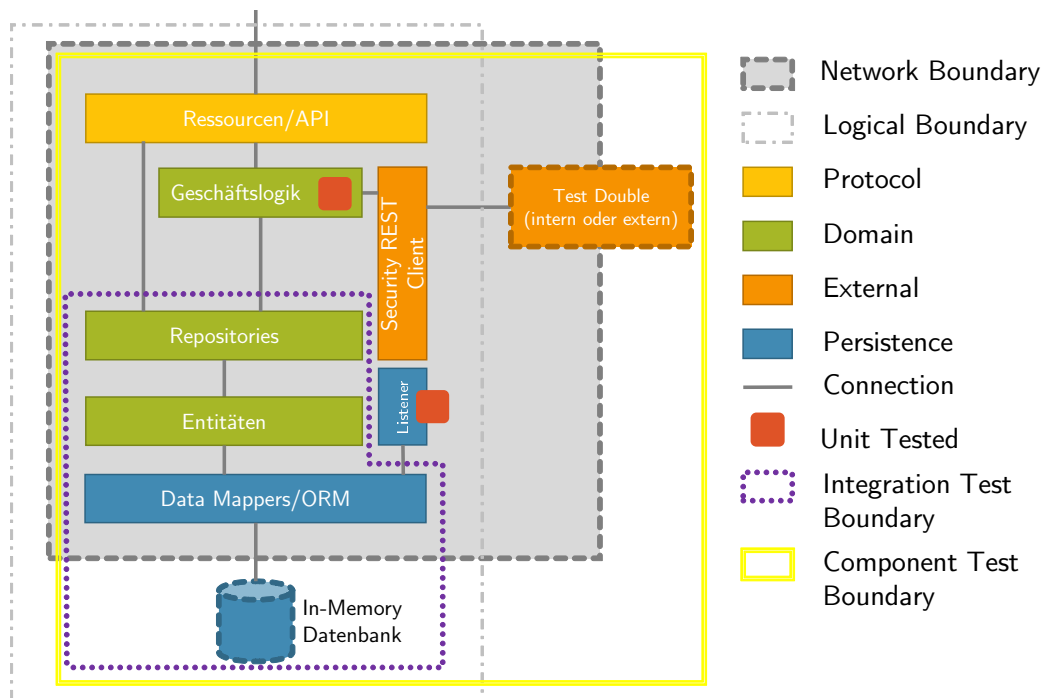


Abbildung 21: Component Testing Scope [Cle14]

Bei Services die komplexere Integrations- oder Bootlogik besitzen, kann es auch sinnvoll sein, gemockte Services durch externe Stubs der benötigten Services zu ersetzen. Dadurch wird die Testlogik zwar zu Teilen in den Test-Harnisch, der für die korrekte Ausführung der Tests verantwortlich ist, geschoben, doch können zusätzliche Fehlerquellen durch die Verwendung von echten Netzwerkanfragen bereits jetzt erkannt werden. Zu beachten ist jedoch, dass sich dadurch auch die Zeit der Testausführung erhöhen kann.[Cle14] Ein Test-Harnisch ist die Zusammenstellung aller für die Ausführung von Tests nötigen Informationen. Dazu zählen auch Konfigurationen, die beispielsweise Stubs und andere Komponenten für die Tests starten.

5.4 Consumer Driven Contract Testing (CDCT)

Durch die Kombination von Unit-, Integration- und Component Testing ist bereits eine hohe Code-Abdeckung eines Microservice erreicht und es ist sichergestellt, dass der Service die Geschäftslogik korrekt implementiert hat. Doch für das Gesamtsystem reicht dies nicht aus. Die Gesamtanwendung entsteht erst durch das Zusammenspiel der verschiedenen Services. Momentan ist nicht durch Tests sichergestellt, dass externe Komponenten die eigenen Anforderungen so erfüllen wie diese gewünscht sind, oder dass alle Services der Domäne einwandfrei zusammenspielen.[Cle14] Um diese Lücken zu schließen bieten Contract Testing und End-to-End Testing Abhilfe.

Da End-to-End-Tests, also Tests bei denen die gesamte Domäne gestartet und getestet

wird, sehr aufwändig sind und nur sehr lange Feedback-Zyklen bieten, steht das Contract Testing als Schicht davor.[Cle14]

Contract Testing basiert auf der Forderung, dass zwischen einem Konsumenten und einem Produzenten ein Vertrag existiert, den der Produzent einzuhalten hat. Dieser Vertrag besteht beispielsweise aus Erwartungen bezüglich In- und Outputs eines Interface. Jeder Konsument hält mit dem Produzenten einen Vertrag, der für seinen Anwendungsfall einzuhalten ist. Wenn am Produzenten Änderungen vorgenommen werden, muss sichergestellt sein, dass alle Verträge ihre Gültigkeit behalten, sodass die abhängigen Konsumenten keine Funktionalität einbüßen müssen.[Cle14]

Bei Microservices besteht das im Vertrag festgehaltene Interface aus der öffentlichen API dieses Service. Jeder Betreiber eines konsumierenden Service schreibt eine Testsuite, die unabhängig voneinander nur die verwendeten Schnittstellen des Produzenten prüft. Im Idealfall werden diese Testsuites dann in der Build Pipeline des produzierenden Service eingebunden, sodass die Entwickler des Produzenten Auswirkungen auf die Konsumenten sofort erkennen können.[Cle14]

Diese Art von Test ist nicht gleichzusetzen mit Component Testing. Es wird nicht die tiefgehende interne Logik eines Service geprüft, sondern lediglich die Erwartungen von In- und Outputs sowie die Einhaltung von Latenz und Durchsatz gegenüber der Realität validiert.[Cle14]

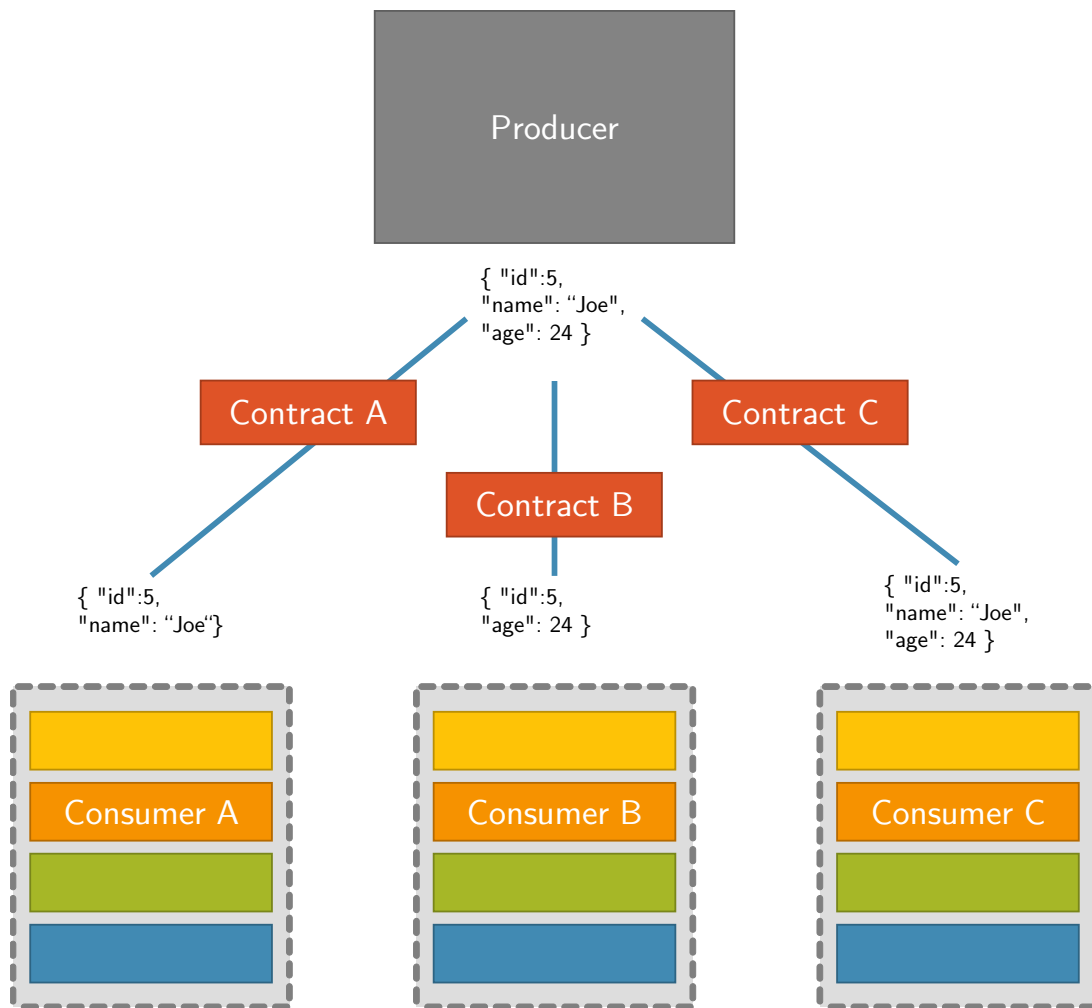


Abbildung 22: Contract Testing [Cle14]

Ein großer Vorteil von dieser Vorgehensweise ist der, dass der produzierende Service viel leichter Änderungen durchführen kann, da er weiß, welche Schnittstellen und Felder von seinen Konsumenten wirklich benötigt werden. So könnte der Produzent aus Abbildung 22 eine Ressource anbieten, die die Felder *id*, *name* und *age* anbietet. Konsument A benötigt nur die *id* und *name*-Felder, somit wird in den Contract Tests zu Vertrag A auch nur die Existenz dieser beiden Felder geprüft. Konsument B benötigt nur das *id* und *age*-Feld, während Konsument C alle Felder benötigt, um korrekt zu funktionieren.[Cle14]

Sollte nun ein weiterer Konsument zur Anwendung hinzukommen, der neben dem Vor- und Nachnamen auch einen Nachnamen benötigt, können die Betreiber des Produzenten sich dazu entscheiden das *name*-Feld als veraltet zu markieren und ein neues Zusammengesetztes Objekt aus Vor- und Nachname anbieten. Durch die Durchführung aller Component Tests wird schnell deutlich, dass die Konsumenten A und C von dieser Änderung betroffen und somit informiert werden müssen. Sobald die beiden Konsumenten an die neue Schnittstelle angepasst sind, kann das alte Feld *name* aus der Ressource entfernt werden.

Laufen dann alle Contract Tests ohne Fehler, ist die Migration auf die neue Schnittstelle erfolgreich.[Cle14]

5.5 End-To-End Testing

Der letzte Schritt zur Sicherstellung, dass alle funktionalen Anforderungen des Systems erfüllt sind, ist das End-to-End Testing. Im Gegensatz zu den vorherigen Testmethoden soll durch diese Testmethode festgestellt werden, dass sämtliche Geschäftsanforderungen vom System gelöst werden, unabhängig von der Architektur der verwendeten Komponenten. [Cle14]

Das Gesamtsystem wird für End-to-End Tests als Black Box gesehen und es werden möglichst viele der Aufgaben des Systems über öffentliche Schnittstellen getestet. Da in Microservice-Architekturen aufgrund ihrer hohen Modularität viele Lücken zwischen den Komponenten entstehen, helfen diese Tests auch die Konfigurationen von etwa verwendeter Infrastruktur wie Firewalls, Proxies und Load-Balancern zu testen.[Cle14]

Verlangt ein System nach Nutzer-Interaktion kann die durch einen Service bereitgestellte GUI durch die Verwendung von etwa Selenium⁷ getestet werden. Dazu werden typische Nutzerabläufe, beispielsweise aus User Stories, aufgenommen und automatisiert ausgeführt, sodass ein Zusammenspiel des Gesamtsystems nötig ist.[Cle14]

Für Systeme die keine Nutzeroberfläche bieten, werden direkt die öffentlichen APIs der Services angesprochen um diese zu manipulieren. Dies geschieht durch die Verwendung eines, möglichst automatischen, HTTP-Clients.[Cle14]

⁷<http://www.seleniumhq.org/>

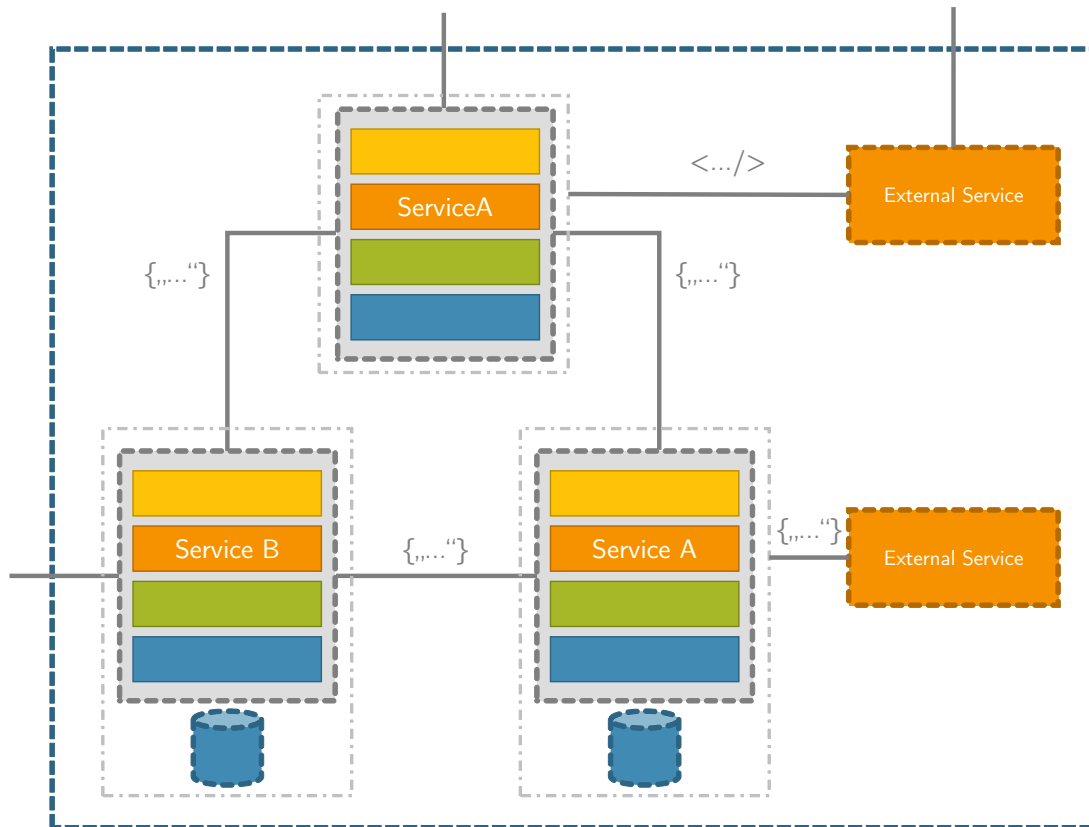


Abbildung 23: End-To-End Testing Scope mit externen Abhängigkeiten [Cle14]

Sollte nicht ein Team komponentenverantwortlich für die Gesamtanwendung sein, werden bei End-to-End Tests auch externe Services mit eingesetzt (siehe Abb. 23). Es gibt allerdings Fälle in denen es sinnvoll sein kann, diese aus dem Scope zu exkludieren. Bei Abhängigkeiten, die nicht einmal mehr in den Händen der Firma selbst, sondern bei Drittanbietern liegen, gibt es viele Faktoren, wie etwa die Stabilität der Systeme des Drittanbieters selbst, die die Testergebnisse für das eigene System verfälschen können. In solchen Situationen ist es Sinnvoll, auch in End-to-End Tests noch Stubs dieser externen Systeme zu verwenden, um sinnvolle Resultate für die eigene Anwendung zu erhalten. Dadurch verliert zwar die Testmethode selbst ein wenig an Vertraulichkeit in Bezug auf die Resultate, jedoch bleiben diese Resultate wenigstens stabil.[Cle14]

End-to-End Tests haben einen großen Nachteil: Sie sind die Tests mit den meisten Faktoren und Komponenten und haben dadurch auch die größte Spanne an möglichen Schwachstellen die zu einem Fehlschlagen der Tests führen können. Dies führt dazu, dass die Laufzeit dieser Tests extrem hoch sein kann und auch das Pflegen der Test Suites ist mit einem großen Arbeitsaufwand verbunden.[Cle14] Daher ist es wichtig beim Schreiben eben dieser auf einige Punkte acht zu geben.

So wenige End-to-End Tests schreiben wie möglich Da ein hoher Vertrauensgrad be-

reits durch die vorhergehenden Teststufen erreicht werden kann, und End-to-End Tests nur noch das Zusammenspiel der bereits durchgängig getesteten Komponenten darstellen soll, sollte nicht zu viel Zeit in eine große Anzahl von Testszenarien gesteckt werden. Am besten ist es, sich ein Zeitbudget zurechtzulegen, welches man bereits ist für die Laufzeit aller End-to-End Tests abzuwarten. Sobald das Zeitbudget durch die Laufzeit überschritten wird, sollten die Tests, die am wenigsten Mehrwert bieten detektiert und gelöscht werden. Das Zeitbudge sollte jedoch in Minuten, nicht in Stunden festgelegt sein.[Cle14]

Auf user stories und Anwendungsfälle fokussieren Wenn man sich beim End-to-End Testing auf reale Anwendungsfälle der Software beschränkt entsteht auch nicht das Problem, dass zu viele Tests dieses Typs geschrieben werden. Es werden nur die Hauptsituationen in denen sich ein Nutzer wiederfindet geprüft, alle anderen Szenarien werden in den meisten Fällen durch die anderen Testmethoden sichergestellt.[Cle14]

Sinnvolle definition des scopes Wie bereits im vorherigen Abschnitt beschrieben, kann es sinnvoll sein externe Services aus dem End-to-End Testscope auszuklammern. Dies kann auch für GUIs von Vorteil sein, sollten diese beispielsweise Ungenauigkeiten in den Testergebnissen verursachen. Wie bereits zuvor erwähnt, tauscht man die Abdeckung der Tests gegen die Zuverlässigkeit der Ergebnisse.[Cle14]

Auf Infrastructure as Code (IaC) zur Wiederholbarkeit von Tests setzen IaC ist eine Herangehensweise die, insbesondere in agilen Entwicklungsteams, für konsistentere Testumgebungen sorgen soll. Wenn Server manuell aufgesetzt und konfiguriert werden, kann dies zu Problemen führen. Es werden bestimmte Versionen von Softwarepaketen installiert, Konfigurationen am System werden vorgenommen etc. Je mehr dieser von Hand getätigten Schritte in der Konfiguration vorkommen, desto eher bilden sich Server, die keinem anderen mehr gleichen. Martin Fowler bezeichnet diese als *Snowflake Server*[Fow12]. Es ist schwer Testumgebungen an die Produktionsumgebungen anzupassen, umso mehr solcher *Schneeflocken* durch das Rechenzentrum treiben. Der Ansatz von IaC geht dahin, diese Konfigurationen in Code zu definieren, sodass die Konfiguration der Test- und Produktionsumgebungen in einer Datei festgehalten ist und diese sogar mit der Anwendung selber in einem VCS abgelegt werden kann. Auch wird es dadurch möglich, Infrastrukturen bei Änderungen automatisch zu starten und Änderungen an mehreren Systemen gleichzeitig durchführen zu lassen.[Fow12]

Beim End-to-End Testing spielt dies eine große Rolle, da durch IaC homogene Laufzeitumgebungen geschaffen werden können, sodass Fehler durch unterschiedlich kon-

figurierte Server und Systeme ausgeschlossen werden können.[Cle14]

Tests datenunabhängig gestalten Anstatt auf bereits existente Datensätze zu vertrauen, sollten die End-to-End Tests selbst die Möglichkeit zur Erzeugung von Daten über die öffentlichen Schnittstellen der Anwendung nutzen, um bei den Tests ständig mit den gleichen Daten zu arbeiten. Wird dies nicht so umgesetzt, kann es zu fehlerhaften Testergebnissen kommen, die jedoch nicht Fehler in der eigentlichen Anwendung als Ursache haben.[Cle14]

Doch kann man End-to-End Testing auch weitaus weniger strukturiert gestalten. An dieser Stelle kommt Monkey Testing ins Spiel. Anstatt Geschäftsanforderungen in End-to-End Tests zu modellieren verfolgt dieser Ansatz die Idee, eine Anwendung durch lauter zufällige Anfragen zu testen, die keiner menschlichen Logik entsprechen. Dabei wird das System für längere Zeit den zufälligen Anfragen ausgesetzt und es werden auftretende Fehler dokumentiert. Der Vorteil dieses Vorgehens liegt natürlich darin, dass sich keine Gedanken über die Testfälle selber gemacht werden müssen, allerdings sollte man diese Art der End-to-End Tests bestenfalls in Kombination mit dem Abbilden von Geschäftslogiken verwenden.

Durch das Aufbrechen einer Anwendung in mehrere, sauber getrennte Services treten zu teilen neue Grenzen auf, die zuvor wahrscheinlich unentdeckt geblieben werden. Diese neuen Grenzen erlauben es auch, Tests weitaus flexibler zu gestalten, als es bei Monolithen möglich wäre.

Hat ein Service eine immens wichtige Aufgabe im Sinne der Geschäftsanforderungen, ist es sinnvoll die volle Bandbreite an möglichen Testmethoden im höchsten Detailgrad auf diesen Service anzusetzen um sein korrektes Verhalten zu garantieren. Wird allerdings ein weiterer Service betrachtet, der weniger wichtige Aufgaben, oder gar experimentelle Funktionen bereitstellt, kann es Sinn machen, die Entwicklungszeit auf andere Gebiete zu fokussieren und nur ein paar der hier vorgestellten Herangehensweisen zu implementieren.[Cle14]

5.6 Code-Generierung

Die Testmethoden sind nun festgelegt und würden bereits für eine Qualitätssteigerung der Anwendung von Nutzen sein. Die gewünschte Zeit- und Kostenersparnis ist an dieser Stelle jedoch noch nicht erreicht. Dafür wird die Generierung benötigt.

Für diese Generierung verwendet it@M Barrakuda. Diese Applikation stellt eine Domain Specific Language (DSL), mit der Microservice-Architekturen modelliert werden können. In dieser DSL lassen sich Services einer Domäne durch Angabe von Entitäten und

Geschäftslogiken darstellen. Diese Nutzereingaben werden anschließend in eine Vielzahl von Templates eingespeist und der Code wird generiert.

Um die oben genannten Testmethoden generieren zu können, müssen Templates erstellt werden können. Dazu müssen die Methoden generierbar sein. Generierbar heißt, dass sich anhand der nachfolgenden Referenzimplementierung Muster in den Source-Files erkennen lassen. Anhand dieser Muster können dann generische Templates erstellt werden. Diese Templates stellen dann die Erweiterung von Barrakuda dar und ermöglichen es dem Endnutzer ohne zusätzlichen Zeitaufwand ein getestetes Gesamtsystem zu generieren.

6 Implementierung

6.1 Referenz-System

6.1.1 Architektur

Das Referenzsystem basiert auf der in Kapitel 3 angesprochenen Architektur. Es wird das Spring-Framework verwendet. 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 HATEOAS⁸ genutzt.

Als Datenbank in der Entwicklungsumgebung kommt eine In-Memory-Datenbank von H2 zum Einsatz. In der Produktion größtenteils Oracles DBMS in der Version 12, aber auch MySQL wird verwendet.

Zur Authentifizierung wird ein Authentifizierungs-Service genutzt, der das städtische LDAP als Nutzerverwaltung verwendet. Die Zurodnung der Rechte auf die Nutzer erfolgt intern im Service. Zu Testzwecken ist auch eine Authentifizierung über in der Datenbank abgelegte Nutzer möglich.

Als Discovery-Service wird Netflix Eureka⁹ verwendet. 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 kommunizieren über den Edge-Service mit den Services der Domäne. Hier kommt Netflix Zuul¹⁰ zur Anwendung. Dieses öffnet nach außen eine einzige Schnittstelle, über

⁸<http://projects.spring.io/spring-hateoas/>

⁹<https://github.com/Netflix/eureka>

¹⁰<https://github.com/Netflix/zuul>

die sowohl eine graphische Nutzeroberfläche aufrufen, als auch die einzelnen Services kontaktiert werden können. Auch ermöglicht das Gateway den Zugriff auf bestimmte Endpunkte zu 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 Continuous Delivery mithilfe eines Container-Frameworks zu ermöglichen.

6.1.2 Beispiel-Anwendung

Als Beispielanwendung wird ein einfacher Online-Shop mit dem Arbeitstitel "Kongo", erst modelliert und später implementiert. 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 searchable "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
20             shopping cart";
21         given productID warehouse.OID;
22         given quantity intMin1;
23     }
24 }
25
26 serviceModel ordering package edu.hm.ba.kongo.shop version 1.0{
27
28     customTimeType dateInPast inThePast;
29
30     entity orderingItem {
```

```

31         cart warehouse.OID "123";
32         orderedOn dateInPast searchable "10.10.2010" ;
33     }
34
35     businessAction orderCart {
36         purpose "Receives a shopping cart to create a new order which can
37             then be payed";
38         given cartID warehouse.OID;
39     }
40
41     businessAction sendInvoice{
42         purpose "Sends the value of the costs of the ordered procuts to
43             an invoicing system";
44         given orderID warehouse.OID;
45     }
46
47     businessAction cancelOrder{
48         purpose "Deletes an order and the associated shopping cart with
49             it 's contents";
50         given orderID warehouse.OID;
51     }
52
53     }
54
55     serviceModel warehouse package edu.hm.ba.kongo.shop version 1.0{
56
57         customTextType OID maxLength=36;
58         customTextType stringMin1 minLength=1;
59         customTextType longText type=long;
60         customNumberType float pointNumber minValue=0;
61         customNumberType int minValue=0;
62
63         entity Product {
64             name stringMin1 searchable mainFeature "Bottle";
65             description longText optional "Great for storing water";
66             price float mainFeature "10.5";
67             quantity int mainFeature "5";
68         }
69     }

```

Das Referenz-System besteht, wie in Abbildung 24 zu sehen, 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. Diese enthält die OID, sowie die gewünschte Anzahl.

Möchte ein Kunde dann eine Bestellung aufgeben, 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 mitgenerierte Authentifizierungsservice dient als Kundenverwaltung.

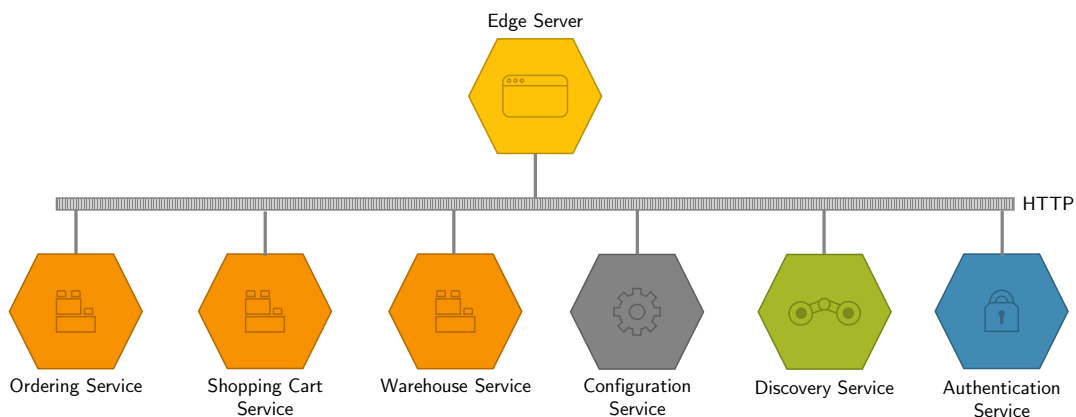


Abbildung 24: Aufbau des Referenzsystems zur Testimplementierung

Wichtig ist, dass im Referenz-System keinerlei Logik zu den modellierten Geschäftsanwendungen implementiert wird, sondern lediglich die im Abschnitt 5 angesprochenen Testmethoden. Dies ist nötig, damit keine zusätzliche Logik, die über die Informationen aus dem Modell hinaus geht, getestet wird. Dies würde nämlich dazu führen, dass sich die implementierten Testmethoden nicht in Templates für Barrakuda übernehmen lassen würden und eine Generierbarkeit nicht mehr gegeben ist.

6.2 Frameworks zum Umsetzen von Test-Strategien

Java als Sprache und Spring als Framework beschränken die Menge der zur Verfügung stehenden Frameworks bereits. Letzten Endes führte dann auch die persönliche Erfahrung im Umgang mit den Frameworks zu der hier genannten Auswahl. Für Unit-, Integration- und Component-Testing wird JUnit unter Zuhilfenahme des Spring Testframeworks *spring-boot-test*¹¹ genutzt. Auch für das CDCT bietet Spring eine eigene Lösung namens *Spring Cloud Contract*¹² die sich leicht und gut in die bestehende Infrastruktur mithilfe von Maven einbinden lässt. Für End-to-End Tests wird Apache JMeter¹³ genutzt. Die Wahl

¹¹<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html>

¹²<https://cloud.spring.io/spring-cloud-contract/>

¹³<http://jmeter.apache.org/>

des richtigen Tools war in diesem Bereich eher schwierig, da es eine sehr große Menge an Möglichkeiten gab. JMeter wurde unter anderem gewählt, weil es bereits in der städtischen IT verwendet wird. Außerdem überzeugt die Kombination aus textueller Beschreibung und ausgesprochen intuitiver grafischer Bedienoberfläche zur Erstellung von Tests. Somit ist nicht nur eine einfache Generierbarkeit gewährleistet, sondern auch eine unkomplizierte Erweiterung des Generats durch einen Entwickler.

6.3 Generativer Ansatz - Was zu beachten ist

Die im Referenzsystem implementierten Tests dienen als Basis für die spätere Verwendung in Barrakuda. Barrakuda verwendet sogenannte Templates, die anhand der gegebenen Informationen aus dem vom Nutzer erstellten Modell befüllt werden. Daher muss bei der Implementierung der Testmethoden unbedingt Rücksicht auf die zur Verfügung stehenden Informationen genommen werden. Informationen, die nicht Modell enthalten sind, können nicht zur Generierung von Tests verwendet werden.

Implementiert man das Referenzsystem, so werden zunächst alle Testmethoden eines Services erstellt. Dieses Vorgehen wird Service für Service wiederholt. Je stärker sich die Implementierungen in den einzelnen Services ähnlich sehen - dies kann man in der Regel an vermehrten copy und paste erkennen - desto wahrscheinlicher ist dieser Code ein Kandidat für die Generierung. Das heißt dieser Test Code muss nicht mehr von Hand implementiert werden, sondern seine Struktur wird in Templates überführt. Dadurch werden die Tests später mit dem Code generiert.

7 Fazit

Insbesondere das Component und Consumer Driven Contract Testing (CDCT) erweisen sich bei Microservice-Architekturen als sehr sinnvoll. Unit- und Integrationstests unterscheiden sich aufgrund der Größe ihres Scopes in Sinn und Zweck nicht von der Verwendung in monolithischen Anwendungen. Aber bereits beim Component Testing ist es bei der Implementierung von sinnvollen Testfällen möglich, bereits Abweichungen in der Definition von nach außen getragenen Schnittstellen zu erkennen. Und das bereits, bevor aufwändige Test Doubles oder gar ganze Infrastrukturen gestartet werden müssen.

Allerdings erweist sich CDCT für den generativen Ansatz als nicht sehr hilfreich. Im Modell fehlen schlichtweg zu viele Informationen über die Kommunikation, die zwischen den Services stattfindet. Somit werden Contracts generiert, die schlichtweg alle möglichen Schnittstellen definieren, auch wenn diese eventuell später gar nicht als Kommunikationsweg genutzt werden. Sinnvoll sind diese allerdings immer noch dafür, die Verwendung von

End-to-End Tests einzuschränken. Die korrekte Funktionalität von Schnittstellen wird bereits in diesem Testschritt gewährleistet, sodass in den End-to-End Tests wirklich nur noch das Zusammenspiel aller Services geprüft werden muss.

Der generative Ansatz wird in jedem Fall zu einer Zeitersparnis führen. Wie in den Abbildungen 25 und 26 zu sehen ist, wird mit den Tests, die generierbar sind bereits eine code coverage von 85%, bzw. 87% erreicht. Und das mit einer vergleichsweise geringen Menge an input-Daten aus dem Modell.




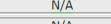

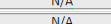
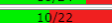
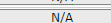
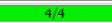
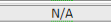


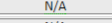
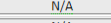
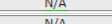
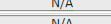
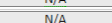
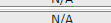
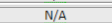
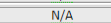
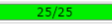


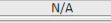

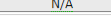


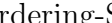

Package /	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	25	85% 	90% 	1,355
edu.hm.ba.kongo.shop.ordering.service	3	88% 	N/A 	1
edu.hm.ba.kongo.shop.ordering.service.gen.businessActionParams	3	100% 	N/A 	1
edu.hm.ba.kongo.shop.ordering.service.gen.controller.businessactions	2	75% 	N/A 	1
edu.hm.ba.kongo.shop.ordering.service.gen.controller.event	1	45% 	N/A 	1
edu.hm.ba.kongo.shop.ordering.service.gen.controller.resource	1	100% 	N/A 	1
edu.hm.ba.kongo.shop.ordering.service.gen.domain	1	100% 	85% 	2,875
edu.hm.ba.kongo.shop.ordering.service.gen.rest	1	N/A 	N/A 	1
edu.hm.ba.kongo.shop.ordering.service.gen.services.businessactions	4	N/A 	N/A 	1
edu.hm.ba.kongo.shop.ordering.service.gen.services.event	1	N/A 	N/A 	1
edu.hm.ba.kongo.shop.ordering.service.gen.services.resource	1	N/A 	N/A 	1
edu.hm.ba.kongo.shop.ordering.service.rest	1	N/A 	N/A 	0
edu.hm.ba.kongo.shop.ordering.service.services.businessactions	4	100% 	100% 	4
edu.hm.ba.kongo.shop.ordering.service.services.event	1	90% 	N/A 	1
edu.hm.ba.kongo.shop.ordering.service.services.resource	1	100% 	N/A 	1

Abbildung 25: Code coverage des Ordering-Service

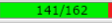


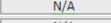

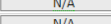

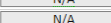
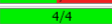
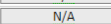

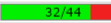
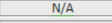
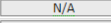
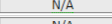
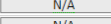
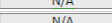
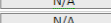
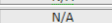
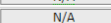
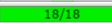


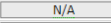

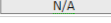


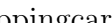

Package /	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	20	87% 	76% 	1,462
edu.hm.ba.kongo.shop.shoppingcart.service	3	88% 	N/A 	1
edu.hm.ba.kongo.shop.shoppingcart.service.gen.businessActionParams	1	100% 	N/A 	1
edu.hm.ba.kongo.shop.shoppingcart.service.gen.controller.businessactions	2	77% 	N/A 	1
edu.hm.ba.kongo.shop.shoppingcart.service.gen.controller.event	1	45% 	N/A 	1
edu.hm.ba.kongo.shop.shoppingcart.service.gen.controller.resource	1	100% 	N/A 	1
edu.hm.ba.kongo.shop.shoppingcart.service.gen.domain	2	96% 	72% 	2,722
edu.hm.ba.kongo.shop.shoppingcart.service.gen.rest	1	N/A 	N/A 	1
edu.hm.ba.kongo.shop.shoppingcart.service.gen.services.businessactions	2	N/A 	N/A 	1
edu.hm.ba.kongo.shop.shoppingcart.service.gen.services.event	1	N/A 	N/A 	1
edu.hm.ba.kongo.shop.shoppingcart.service.gen.services.resource	1	N/A 	N/A 	1
edu.hm.ba.kongo.shop.shoppingcart.service.rest	1	N/A 	N/A 	0
edu.hm.ba.kongo.shop.shoppingcart.service.services.businessactions	2	100% 	100% 	3,5
edu.hm.ba.kongo.shop.shoppingcart.service.services.event	1	90% 	N/A 	1
edu.hm.ba.kongo.shop.shoppingcart.service.services.resource	1	100% 	N/A 	1

Abbildung 26: Code coverage des Shoppingcart-Service

Als Erweiterung wäre es aber, wie bereits erwähnt, sinnvoll, dass Kommunikationsschnittstellen zwischen modellierten Services in das Modell mit aufzunehmen, um sinnvollere Testfälle insbesondere in den CDCT und End-to-End Tests generieren zu können.

Doch für einen höheren Informationsgehalt das Model von Barrakuda zu verkomplizieren ist eventuell auch keine gute Lösung, da insbesondere die schnelle und einfache Modellierung aktuell den Reiz der Anwendung ausmacht. Und Kommunikationswege anschaulich und überschaubar in Textform zu modellieren ist ebenfalls nicht leicht umzusetzen.

Zu einem Großteil der Zeitersparnis führt auch das Grundgerüst, was mit den im Rahmen dieser Arbeit entwickelten Tests bereitgestellt wird. Bei it@M wird gerade erst mit

der produktiven Verwendung von Microservice-Architekturen begonnen. Ein Großteil der Entwickler muss sich mit teilweise völlig neuen Frameworks auseinandersetzen, neue Denkweisen bei der Lösung von Problemen müssen bedacht werden. Sind in den generierten Systemen nun bereits grundlegende Tests implementiert, und die Strategien hinter diesen erläutert, erlaubt dies den Entwicklern sich auf das Erstellen von Testfällen zu konzentrieren, ohne sich dabei zunächst Gedanken um die Verwendungen des Frameworks machen zu müssen oder sich mit Konfigurationen zu beschäftigen. Dieser schnelle und unkomplizierte Einstieg kann die Motivation zum schreiben von Tests erhöhen.

Auch sorgt die Generierung für weitere Standardisierung in den Systemen. Wurden bereits Vorkehrungen im Generat getroffen, die das automatische Ausführen der Testmethoden ermöglichen, werden diese auch mit hoher Wahrscheinlichkeit genutzt werden, sodass der CI-Prozess verbessert wird. Ebenso wie es bereits durch die Generierung von Docker-Files und dem damit entstehenden Interesse an Container-Lösungen der Fall war.

Abschließend lässt sich sagen, dass der generative Ansatz zur Testerstellung in Microservice-Architekturen in diesem Anwendungsfall sinnvoll ist. Dies liegt aber vor allem darin, dass bereits eine domänenspezifische Sprache, sowie der dazugehörige Generator vorliegt, und die im Rahmen dieser Arbeit erarbeiteten Testmethoden nur eine Erweiterung darstellen. Die Möglichkeit eine eigenständige Applikation zu entwickeln, die für die reine Testgenerierung zuständig ist, besteht, doch ist dies nahezu unmöglich (aufgrund der vielen verfügbaren Frameworks und Sprachen zur Umsetzung einer Microservice-Architektur). Außerdem würde die Verwendung einer solchen Software hohe Zeitaufwände kosten, um bestehende Infrastrukturen neu zu modellieren, um Tests zu generieren. Währenddessen ist die Erweiterung von Barrakuda ein zusätzliches Feature, welches Anwendern und den Entwicklern der Anwendungen hoffentlich nicht nur zeitliche Vorteile verschaffen kann.

8 Quellenverzeichnis

- [Cha05] CHARETTE, Robert N.: Why Software Fails. (2005). <http://spectrum.ieee.org/computing/software/why-software-fails>
- [Cle14] CLEMON, Toby: Testing Strategies in a Microservice Architecture. (2014). <http://martinfowler.com/articles/microservice-testing/>. – Zuletzt Abgerufen am 01.11.2016
- [Deg15] DEGGES, Randall: What the Heck is OAuth? (2015). <https://stormpath.com/blog/what-the-heck-is-oauth>. – Zuletzt Abgerufen am 24.01.2017
- [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
- [Eri12a] ERIKSSON, Ulf: Functional vs Non Functional Requirements. (2012). <http://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/>. – Zuletzt Abgerufen am 26.01.2017
- [Eri12b] ERIKSSON, Ulf: Functional vs Non Functional Testing. (2012). <http://reqtest.com/testing-blog/functional-vs-non-functional-testing/>. – Zuletzt Abgerufen am 26.01.2017
- [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
- [Fow06] FOWLER, Martin: TestDouble. (2006). <https://www.martinfowler.com/bliki/TestDouble.html>. – Zuletzt Abgerufen am 01.02.2017
- [Fow12] FOWLER, Martin: SnowflakeServer. (2012). <https://martinfowler.com/bliki/SnowflakeServer.html>. – Zuletzt Abgerufen am 27.01.2017
- [Fow15] FOWLER, Martin: Microservices Trade-Offs. (2015). <https://martinfowler.com/articles/microservice-trade-offs.html>. – Zuletzt Abgerufen am 24.01.2017
- [Gie16] GIERKE, Oliver: DDD & REST - Domain-Driven APIs for the web. (2016). <https://speakerdeck.com/olivergierke/ddd-and-rest-domain-driven-apis-for-the-web-3?slide=65>. – Zuletzt Abgerufen am 31.01.2017
- [Hag08] HAGEN, Stefan: Das „Triple Constraint“ im Projektmanagement. (2008). http://pm-blog.com/2008/08/04/triple_constraint_magisches_dreieck/. – Zuletzt Abgerufen am 10.01.2017
- [HM03] HEATT, Edward ; MEE, Rob: Repository. (2003). <https://www.martinfowler.com/eaCatalog/repository.html>. – Zuletzt Abgerufen am 01.02.2017

- [Hof14] HOFF, Todd: The Great Microservices Vs Monolithic Apps Twitter Melee. (2014). <http://highscalability.com/blog/2014/7/28/the-great-microservices-vs-monolithic-apps-twitter-melee.html>. – Zuletzt Abgerufen am 24.01.2017
- [Hoh04] HOHPE, Gregor: Enterprise Integration Styles. (2004). <http://www.informit.com/articles/article.aspx?p=169483&seqNum=3>. – Zuletzt Abgerufen am 13.01.2017
- [Inf16] INFLECTRA: Testing Methodologies. (2016). <https://www.inflectra.com/Ideas/Topic/Testing-Methodologies.aspx>. – Zuletzt Abgerufen am 26.01.2017
- [Krz12] KRZYZANOWSKI, Paul: Remote Procedure Calls. (2012). <https://www.cs.rutgers.edu/~pxk/416/notes/17-rpc.html>. – Zuletzt Abgerufen am 03.02.2017
- [Kur16] KURZ, Martin: *Verbesserung des Softwareentwicklungsprozesses der Landeshauptstadt Muenchen durch modellgetriebene Softwareentwicklung*, Hochschule München, Diplomarbeit, 2016
- [Mog16] MOGOSANU, Mike: DDD Decoded - The Aggregate and Aggregate Root Explained. (2016). <http://blog.sapiensworks.com/post/2016/07/14/DDD-Aggregate-Decoded-1>. – Zuletzt Abgerufen am 31.01.2017
- [New15] NEWMAN, Sam: *Building Microservices*. O'Reilly, 2015. – ISBN 9781491950357
- [Pan99] PAN, Jiantao: Software Testing. (1999). https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/. – Zuletzt Abgerufen am 26.01.2017
- [Ric14] RICHARDSON, Chris: Pattern: Monolithic Architecture. (2014). <http://microservices.io/patterns/monolithic.html>. – Zuletzt Abgerufen am 16.01.2017
- [Ric16] RICHARDSON, Chris: Choosing a Microservices Deployment Strategy. (2016). <https://www.nginx.com/blog/deploying-microservices/>. – Zuletzt Abgerufen am 08.02.2017
- [Sam10] SAMSUNG: Anyframe Spring REST Plugin. (2010). <http://dev.anyframejava.org/docs/en/anyframe/plugin/springrest/1.0.2/reference/html/ch02.html>. – Zuletzt Abgerufen am 25.01.2017
- [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
- [Wig12] WIGGINS, Adam: The Twelve-Factor App - Config. (2012). <https://12factor.net/config>. – Zuletzt Abgerufen am 06.02.2017
- [Wol15a] WOLFF, Eberhard: *Microservices: Grundlagen flexibler Softwarearchitekturen*. Dpunkt.Verlag GmbH, 2015 <https://books.google.de/books?id=SrcswEACAAJ>. – ISBN 9783864903137
- [Wol15b] WOLFF, Eberhard: REST vs. Messaging For Microservices. (2015). <http://de.slideshare.net/ewolff/rest-vs-messaging-for-microservices>. – Zuletzt Abgerufen am 02.02.2017