



**Fakultät für Informatik und Mathematik 07**

# **Bacheloararbeit**

über das Thema

<sup>5</sup> **Generative Testerstellung für Microservice-Architekturen**

**Autor:** Fabian Wilms  
holtkoet@hm.edu

**Prüfer:** Prof. Dr. Ulrike Hammerschall

**Abgabedatum:** xx.xx.2017

# I Kurzfassung

kurzfassung

## Abstract

Das ganze auf Englisch.

## II Inhaltsverzeichnis

I	Kurzfassung	I
II	Inhaltsverzeichnis	II
III	Abbildungsverzeichnis	III
5 IV	Tabellenverzeichnis	III
V	Listing-Verzeichnis	III
VI	Abkürzungsverzeichnis	III
1	Einführung und Motivation	1
2	Architekturvergleich Microservice und Monolith	4
10 2.1	Microservice Architektur . . . . .	5
2.1.1	Kommunikation zwischen Services . . . . .	6
2.2	Monolithische Architektur . . . . .	11
2.3	Vorteile von Microservices gegenüber Monolithen . . . . .	13
2.4	Nachteile von Microservices gegenüber Monolithen . . . . .	14
15 3	Vorgegebene Architektur der Landeshauptstadt München (LHM)	15
4	Software Testen	18
4.1	Bekannte Testmethoden . . . . .	18
4.2	Testen von Microservices . . . . .	20
4.2.1	Sinnvolle Teststrategien für den generativen Ansatz . . . . .	25
20 4.2.2	Frameworks zum Umsetzen von Test-Strategien . . . . .	25
5	Anforderungen an generierte Tests	25
5.1	Benötigte Daten . . . . .	25
5.2	Notwendige Änderungen/Erweiterungen von Barrakuda . . . . .	25
6	Implementierung in Barrakuda	25
25 6.1	Referenz-System . . . . .	25
6.1.1	Architektur . . . . .	25
6.1.2	Beispiel-Anwendung . . . . .	26
6.1.3	Komponenten und Aufbau . . . . .	29
6.1.4	Implementierung des Systems . . . . .	29
30 6.1.5	Implementierung der Tests . . . . .	29
6.2	Übernahme der Referenz-Implementierung in Barrakuda-Templates . . . .	29
7	Fazit	29
8	Quellenverzeichnis	29

<b>Anhang</b>	<b>I</b>
<b>A Code-Fragmente</b>	<b>I</b>

### III Abbildungsverzeichnis

	Abb. 1	Magisches Dreieck des Projektmanagements[Hag08] . . . . .	1
5	Abb. 2	SQS Report Costs of Defect Correction [SQS] . . . . .	2
	Abb. 3	Logische Unterscheidung zwischen Monolithischer und Microservice Architektur . . . . .	4
	Abb. 4	Ablauf eines Remote Procedure Call (RPC) . . . . .	7
	Abb. 5	Beispiel einer REST-Konformen Anwendung[Sam10] . . . . .	9
10	Abb. 6	Auslastungsproblem bei der Skalierung von Monolithen . . . . .	13
	Abb. 7	Architekturvorgabe it@M . . . . .	16
	Abb. 8	Kommunikation innerhalb der Architektur . . . . .	17
	Abb. 9	Aufbau eines Service innerhalb der it@M-Architektur . . . . .	18
	Abb. 10	Funktionale und nicht-funktionale Testmethoden . . . . .	19
15	Abb. 11	Umfang der verschiedenen Testmethoden . . . . .	20
	Abb. 12	Unit Testing Scope [Cle14] . . . . .	21
	Abb. 13	Integration Testing Scope [Cle14] . . . . .	22
	Abb. 14	Component Testing Scope [Cle14] . . . . .	23
	Abb. 15	Contract Testing [Cle14] . . . . .	24
20	Abb. 16	End-To-End Testing Scope [Cle14] . . . . .	25

### IV Tabellenverzeichnis

### V Listing-Verzeichnis

code/customer.json . . . . .	10
../ReferenceSystem/.mdsd/referencesystem.barrakuda . . . . .	26

### 25 VI Abkürzungsverzeichnis

<b>LHM</b>	Landeshauptstadt München
<b>LDAP</b>	Lightweight Directory Access Protocol
<b>HATEOAS</b>	Hypermedia as the engine of application state
<b>RPC</b>	Remote Procedure Call
30 <b>REST</b>	Representational State Transfer
<b>API</b>	Application Programming Interface
<b>ORM</b>	Object-Relational Mapping

# 1 Einführung und Motivation

IT nimmt sowohl im privaten als auch geschäftlichen Alltag eine immer größere Rolle ein. Die Übernahme von Bereichen, die ehemals als nicht durch Computer austauschbar erachtet wurden, schreitet immer weiter fort. Doch dadurch steigen nicht nur bestehende  
5 Anforderungen an Software, sondern es entstehen auch neue Kriterien an die Qualität. Sobald Türklingeln, Alarmanlagen und Schließanlagen *smart* werden, ist die Fehlertoleranz gleich null. Es ist bereits zu schwerwiegenden Unfällen, wie etwas Flugzeugabstürzen bekommen.[Pan99] Ganz abgesehen davon steigt auch die Komplexität von modernen Software-Systemen immens an, auch weil dies von den Nutzern der Anwendungen durch  
10 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  
15 Project Management Triangle.

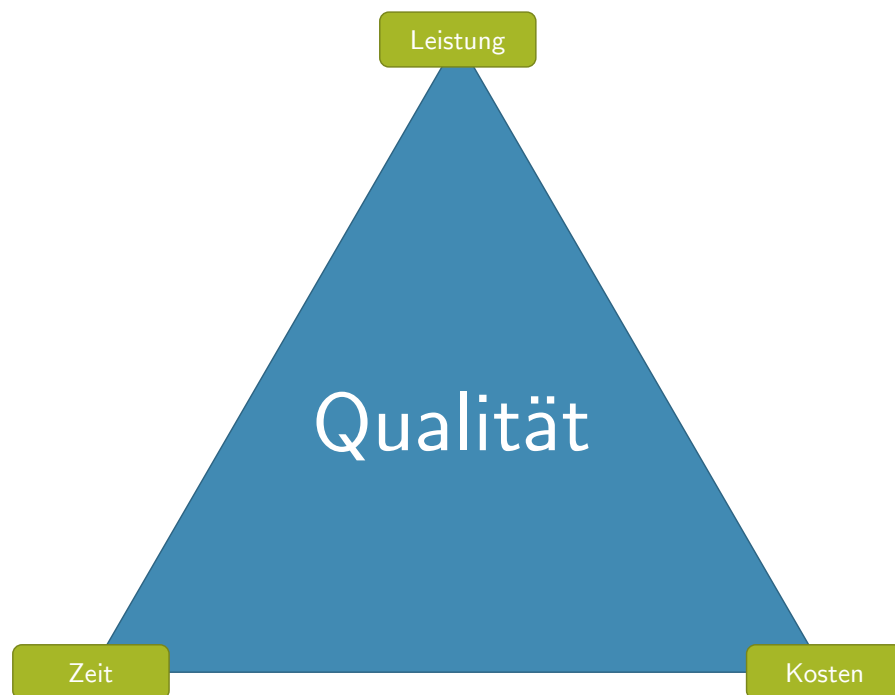


Abbildung 1: Magisches Dreieck des Projektmanagements[Hag08]

Dieses besagt, dass die Qualität eines Projekts durch die drei Faktoren der Leistung,

Kosten und Zeit beeinflusst wird. 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. Somit werden zum Ausgleich die Kosten erhöht.

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

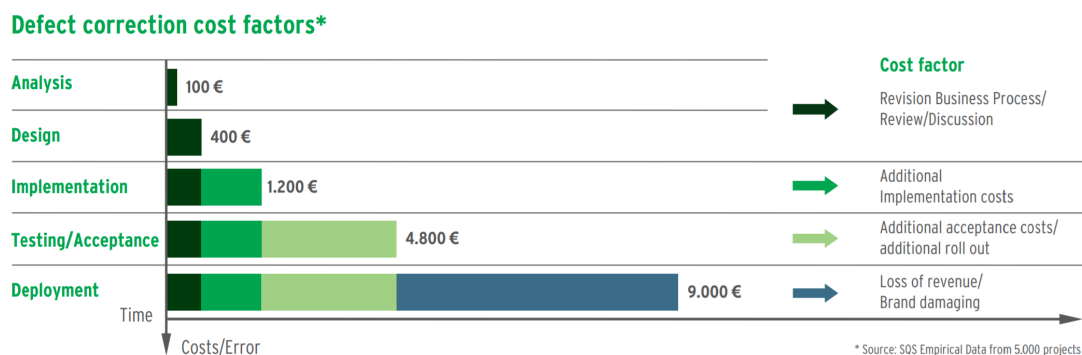


Abbildung 2: SQS Report Costs of Defect Correction [SQS]

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

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

*„If the software coders don't catch their omission until final system testing—or worse, until after the system has been rolled out—the costs incurred to correct the error will likely be many times greater than if they'd caught the mistake*

*while they were still working on the initial [...] process.*“

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

Bei it@M, dem IT-Dienstleister der Landeshauptstadt München, ist eine hohe Testabdeckung daher Teil der Definition of Done [citation needed] von Softwareprojekten. Im Kommunalen Umfeld sind die zeitlichen Restriktionen noch einmal stärker zu gewich-  
10 ten 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.

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

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

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

Schließlich werden bekannte Methoden zum Testen von Software analysiert (Kapitel 4.1)  
30 und es werden alternative Methoden, die besonders im Bereich der Microservices anzutreffen sind, ebenfalls untersucht (Kapitel 4.2). Diese werden dann in Hinblick der Umsetzungsmöglichkeit im generativen Ansatz geprüft (Kapitel 4.2.1) und es werden Frameworks die zur Implementierung genutzt werden sollen definiert (Kapitel 4.2.2) .

---

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

Schließlich sollen Anforderungen, die an die zu generierenden Tests gestellt werden (Kapitel 5), festgehalten werden. Diese finden dann im letzten Abschnitt, der Implementierung, Beachtung (Kapitel 6).

## 2 Architekturvergleich Microservice und Monolith

- 5 Architekturen gibt es in der Welt der Softwareentwicklung viele. Eine der momentan bekanntesten Neuerungen an den Whiteboards der IT-Architekten ist die "MicroserviceÄrchitektur. Viele kleine Services die im Konglomerat für ein gemeinsames Ziel zusammenarbeiten. Vorbei sein soll die Zeit der Monolithischen Softwaresysteme, bei denen die Gesamtlogik in einer einzigen Anwendung gekapselt ist.

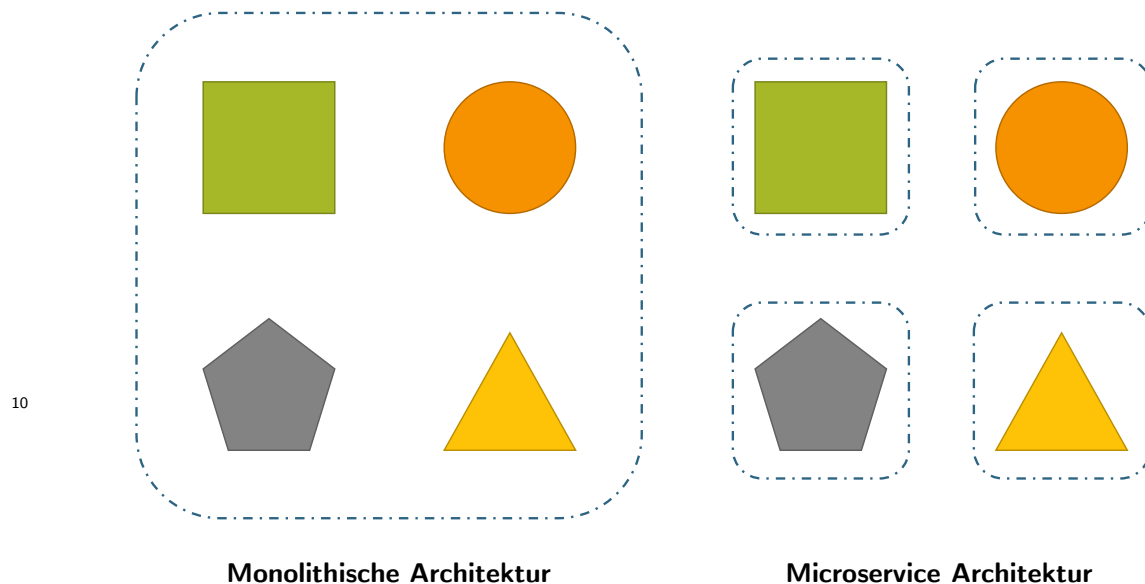


Abbildung 3: Logische Unterscheidung zwischen Monolithischer und Microservice Architektur

- 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
- 15 Detail betrachten.



## 2.1 Microservice Architektur

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

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

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

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

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

Sie lassen sich gut am Beispiel eines Online-Shops zeigen. Wenn ein Online-Shop geplant wird ist einer der zentralen Bestandteile die Lagerverwaltung. Die Daten der Lagerverwaltung werden herangezogen um dem potentiellen Kunden des Shops anzuzeigen welche und wie viele Produkte verfügbar sind. Doch möchte ein Kunde nun ein Produkt bestellen  
20 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  
25 Kompetenz dieses Systems nichts mehr zu tun. Somit wird eine weitere Kontextgrenze erstellt, nämlich die der Kundenverwaltung.

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

S.68]

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

### 2.1.1 Kommunikation zwischen Services

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

#### 2.1.1.1 Geteilte Datenbanken sind eine Möglichkeit zur Servicekommunikation [New15,

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

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

#### 2.1.1.2 Remote Procedure Calls (RPCs) sind eine weitere bekannte Kommunikati-

onsmöglichkeit. 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 eine Anfrage an einen Server sendet. In dieser Anfrage ist der Name oder die ID einer Methode, sowie die zugehörigen Parameter enthalten. Der empfangende Server führt die gewünschte Methode bei sich aus und liefert dem Client als Antwort den Rückgabewert der Methode. Ein Vorteil von RPC ist, dass es sehr einfach und schnell möglich ist Methoden der Services für Clients und

andere Services Verfügbar zu machen. Über die Kommunikation muss man sich nahezu keine Gedanken machen[New15, S.91].

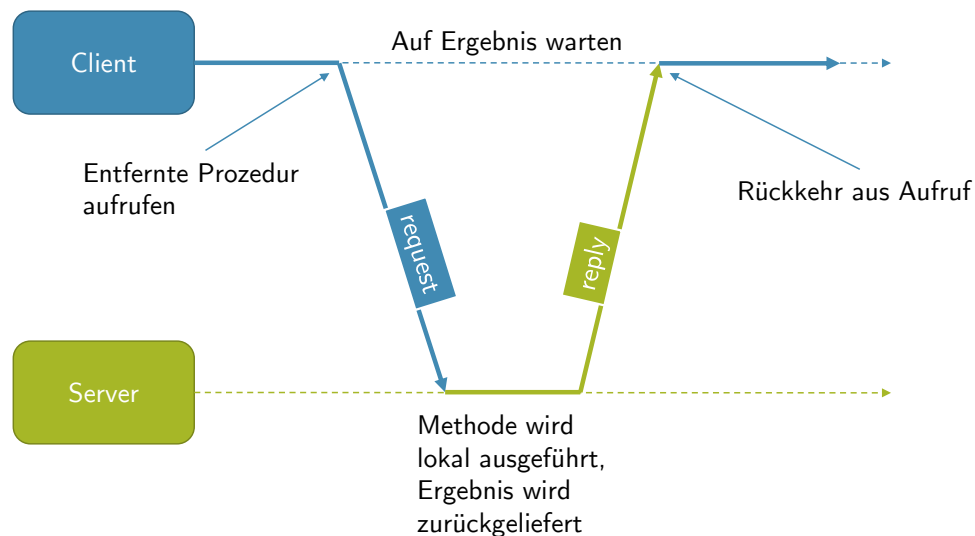


Abbildung 4: Ablauf eines RPC

- 5 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].
- 10 Die Verwendung von RPC macht die Weiterentwicklung von Systemen nicht einfacher: Ein Beispiel anhand einer Schnittstelle zur Instanziierung eines Kunden. Zusätzlich zur Erstellung eines Kunden mithilfe seines Namen und einer E-Mail Adresse soll es nun eine Möglichkeit geben diesen nur mithilfe seiner Mail-Adresse zu erstellen. Das reine hinzufügen einer neuen Methode in einem Interface löst das Problem in diesem Fall nicht.
- 15 Im schlimmsten Fall benötigen alle Clients die diesen Service ansprechen die neuen Stubs und müssen allesamt neu Bereitgestellt werden[New15, S.94]. Und das bereits bei einer so kleinen Änderung.

**2.1.1.3 Representational State Transfer (REST)** ist eine der meistgenutzten Programmierparadigmen für 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 6 Eigenschaften, die erfüllt sein müssen um eine REST-Konforme Anwendung zu schreiben.

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

**Identification of resources** Jede, über eine URI erreichbare Information ist eine 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, einfach gesagt, Informationen Links zu anderen Informationen enthalten.

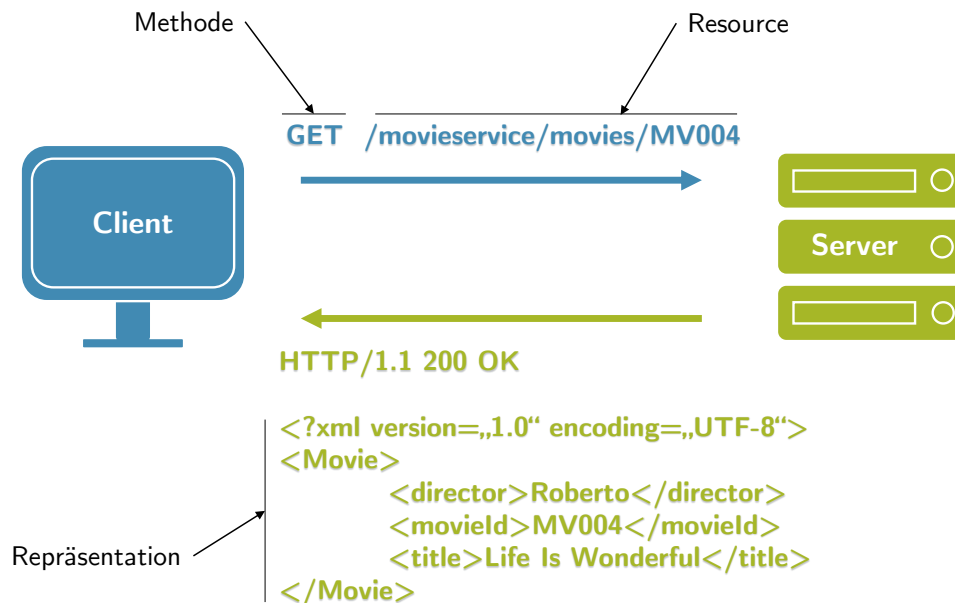


Abbildung 5: Beispiel einer REST-Konformen Anwendung[Sam10]

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

Die Verwendung von REST bietet viele Möglichkeiten die lose Kopplung zwischen Services zu ermöglichen. Dazu zählt unter anderem HATEOAS. Um bei dem Kundenbeispiel aus 2.1.1.2 zu bleiben: Wird die Information eines Kunden abgerufen, kann das Kundenobjekt

zusätzlich zu den Kundendaten ein Feld enthalten welches zur Bestellliste dieses Kund zeigt.

---

```
1  {  
5  2    "name": "Hans Meier",  
3    "mail": "hansmei@mail.de",  
4    "links": [{  
5        "rel": "orders",  
6        "href": "https://mycompany.com/orders/12341232/"  
10 7    }]  
8  }
```

---

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

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

**2.1.1.5 Authorisierung** Beim Thema der Clientauthorisierung von Web-APIs ist OAuth2 weit verbreitet [citation needed]. Wichtig ist hierbei die Unterscheidung zwischen Authorisierung 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]

OAuth2 bietet dazu sogenannte *Grant Types* an. Vier an der Zahl, und alle für unterschiedliche Anwendungsfälle.

**Authorization Code** Wird verwendet, um den Login über einen Drittanbieter auf einer Seite zu ermöglichen. Zum Beispiel über Twitter, Facebook oder Google. Der Nutzer wird beim versuch sich einzuloggen auf die Seite des OAuth2-Providers weitergeleitet um dort seine Identität zu bestätigen. Anschließend wird er zurückgeleitet auf die

Ursprüngliche Website. Diese erhält vom Provider einen *authorization code* durch den die Authentifizierung sichergestellt ist. Mit diesem *authorization code* kann nun eine Anfrage an den Provider gestellt werden um einen *access token* zu erhalten, mit dem dann auch Daten des Nutzers angefragt werden können.[Deg15]

- 5 **Implicit** Dieser *Grant Type* funktioniert auf eine ähnliche Weise wie der des *Authorization Codes*, jedoch wird kein *authorization code* zurückgeliefert, mit dem anschließend ein *access token* angefordert werden muss. Stattdessen wird direkt ein *access token* zurückgeliefert.[Deg15] Dieser Grant Type sollte jedoch nur in speziellen Fällen (wie zum Beispiel bei reinen Client-Seitigen Anwendungen, die nur kurzzeitig Zugriff über einen *access token* benötigen) verwendet werden, da er im Gegensatz zum *Authorization Code* unsicherer ist.

- Password** Bei Verwendung eines eigenen OAuth2-Servers bietet dieser Grant Type die Möglichkeit einen Nutzer über eine Nutzernamen und Passwort Kombination zu Authentifizieren und anschließend zu Autorisieren. Bei einer Anfrage erhält die anfragende Anwendung, vorausgesetzt der Nutzer konnte authentifiziert werden, den *access token*. [Deg15]

- Client credentials** Die letzte Möglichkeit findet nur bei rein automatischen Anwendungen ihren Einsatzzweck. Beispielsweise für Hintergrundprozesse die Datenanalyse betreiben o.ä. Solche Anwendungen verwenden den *Client Credentials* grant type um mithilfe einer client id und eines secrets ein *access token* zu erhalten.[Deg15]

## 2.2 Monolithische Architektur

- Monolithische Software stellt das Gegenteil zu Microservices dar. Eine Monolithische Anwendung vereint die Gesamtlogik in einer einzigen Laufzeitumgebung und ist dadurch meist sehr groß. Die Aufteilung der Logik innerhalb der Anwendung findet nur auf Entwickler-Ebene statt. 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.

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

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

- 5 Doch ab einem gewissen Entwicklungsstand nehmen die Probleme überhand. Viele Leute arbeiten an einer Code-Basis die stetig wächst. Wenn über den Projektzeitraum neue Entwickler am Projekt teilnehmen können diese zunächst leicht überfordert sein, da die anfängliche Modularität dadurch, dass sie nicht hart vorgeschrieben ist, stetig abnimmt. Dies führt auch dazu, dass bei Änderungen an großen Systemen nicht mehr direkt klar  
10 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.

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

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

- Auch wenn die Skalierung in der Theorie sehr einfach funktionieren sollte, tut sie das in der Praxis meist nicht. Das Problem: Monolithen lassen sich nur eindimensional skalieren. Auf eine höhere Anzahl von Anfragen kann also durch das starten einer weiteren  
25 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.



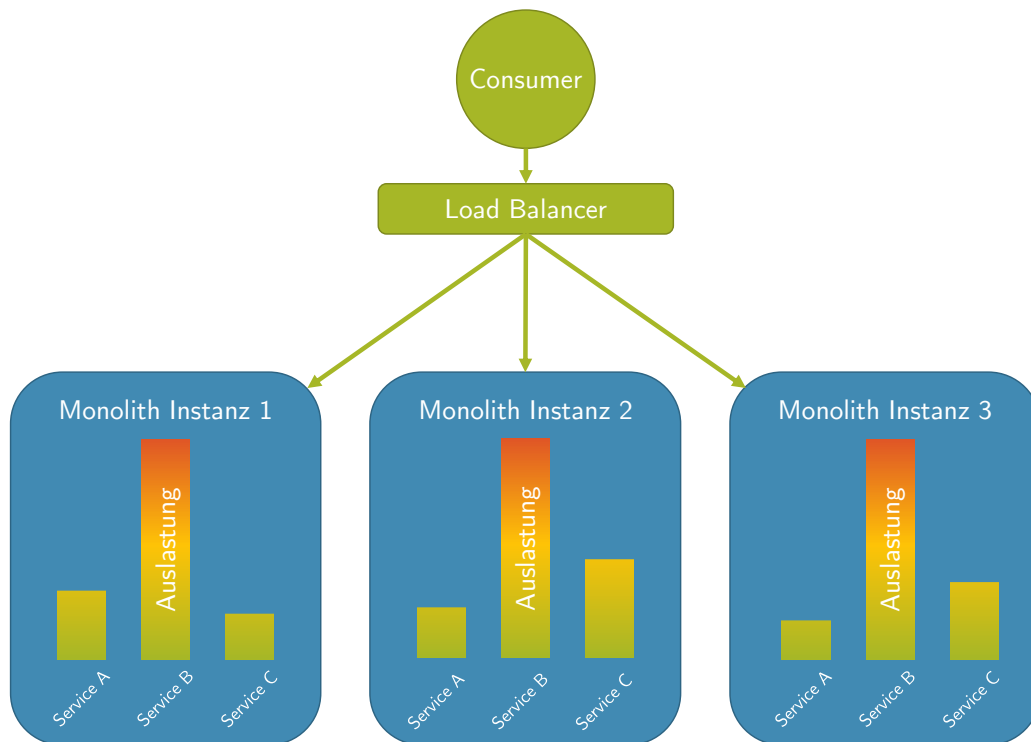


Abbildung 6: Auslastungsproblem bei der Skalierung von Monolithen

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

## 2.3 Vorteile von Microservices gegenüber Monolithen

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

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

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

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

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

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

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

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

## 2.4 Nachteile von Microservices gegenüber Monolithen

Doch gibt es wie bei jeder Architektur nicht nur Vorteile. Ansonsten würde es auch keine Diskussionen und Neuerungen auf diesem Gebiet geben.

Eine der deutlichsten Probleme ist die der 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].

Die Komplexität der laufenden Services auf der Infrastruktur ist ein weiterer negativer Aspekt. Auch wenn die Komponenten einzeln ausgetauscht und aktualisiert werden können, muss häufig das Gesamtsystem in einer bestimmten Abfolge gestartet werden um eine reibungslose Kommunikation zu ermöglichen. Und durch die Möglichkeit schnell Neuerungen in die Produktion zu bringen, werden die das Deployment steuernden Teams viel Arbeit haben.[Fow15]

Als letzter Punkt ist die *Eventual Consistency* anzusprechen. 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.

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, aber dennoch miteinander kommunizieren um gemeinsam das Produkt des Betriebssystems zu bilden.[Hof14]

Auch sind Monolithische Architekturen nicht gleich schlechter oder unnutzbar. Bestes aktuelles Beispiel dafür ist die Firma etsy<sup>2</sup>, die ihren Online-Flohmarkt in Form einer Monolithischen Anwendung entwickeln und erfolgreich betreiben.

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. 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)

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. Sowie einen Discovery-Service, der allen beteiligten Services einen Anlaufpunkt zur Service-Discovery bietet. Zusätzlich existiert ein Configuration Service der die zentrale Steuerung von gemeinsamen Einstellungen erlaubt.

Zusätzlich zur Grundstruktur kann es in einer Anwendung mehrere Services geben, die alle innerhalb ihrer Kontextgrenzen Aufgaben erfüllen. Es wird ein ORM-Mapper zur Nutzung einer relationalen Datenbank genutzt. Des weiteren gibt es einen Edge Server. Dieser funktioniert 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.

---

<sup>2</sup><https://www.etsy.com/about/?ref=ftr>

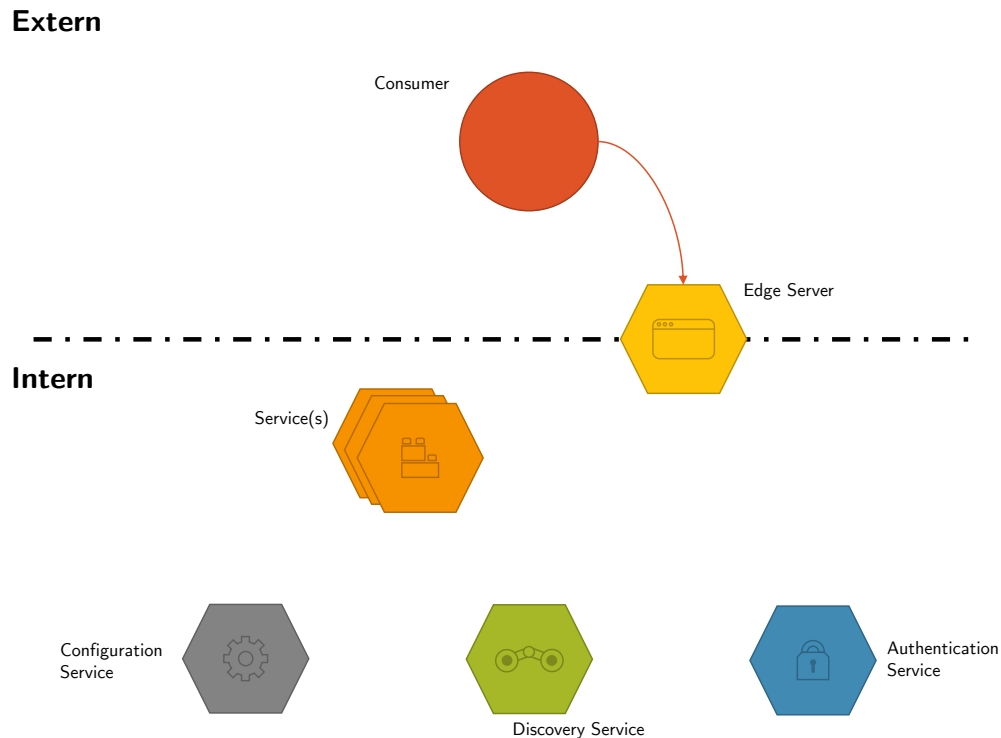


Abbildung 7: Architekturvorgabe it@M

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

- 5 Der Konsument der Anwendung, sei es ein Nutzer oder eine weitere Software, kommuniziert ausschließlich über das Gateway mit den internen Komponenten. Wird eine Anfrage gestellt muss zunächst die Authentifizierung sichergestellt sein. Anschließend wird über den Discovery Service der zur Bearbeitung der Anfrage benötigte Service gesucht und die Anfrage an diesen gestellt. Bei allen gesicherten Endpunkten innerhalb der Anwendung
- 10 findet immer eine Absprache mit dem Authentifizierungsservice statt, um auf eine Änderung der Rechte eines Nutzers sofort reagieren zu können.

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

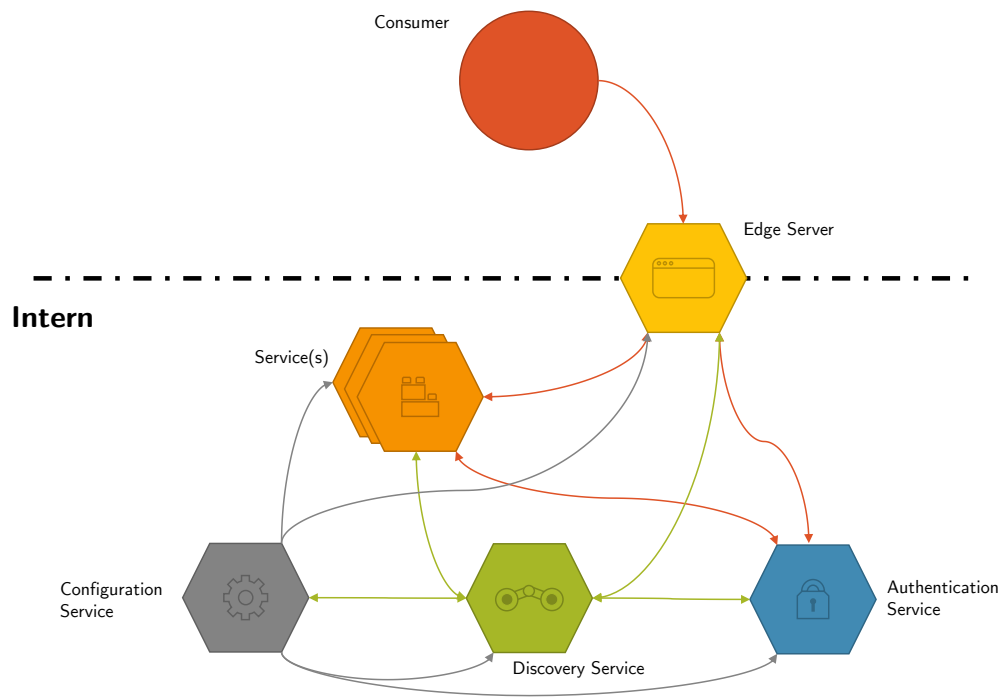
**Extern**

Abbildung 8: Kommunikation innerhalb der Architektur

Jeder Service der Anwendung hat intern eine gleiche Aufteilung. Nach außen werden, REST-konform, nur die Ressourcen über **URI!s (URI!s)** angeboten. Diese Ressourcen  
 5 bestehen entweder aus Entitäten oder Geschäftsanwendungen. Entitäten werden über sogenannte Repositories als Ressource bereitgestellt und über eine ORM-Bibliothek in einer Datenbank persistiert. Zum zusätzlichen Eingriff durch Entwickler können, kann auf verschiedene Events mithilfe von Listnern reagiert werden. Zur Kommunikation mit anderen Services existiert ein *Security REST Client*, der die Kommunikation mit anderen Services  
 10 durch Verwendung des Discovery und Authentication Service so einfach wie möglich machen soll.

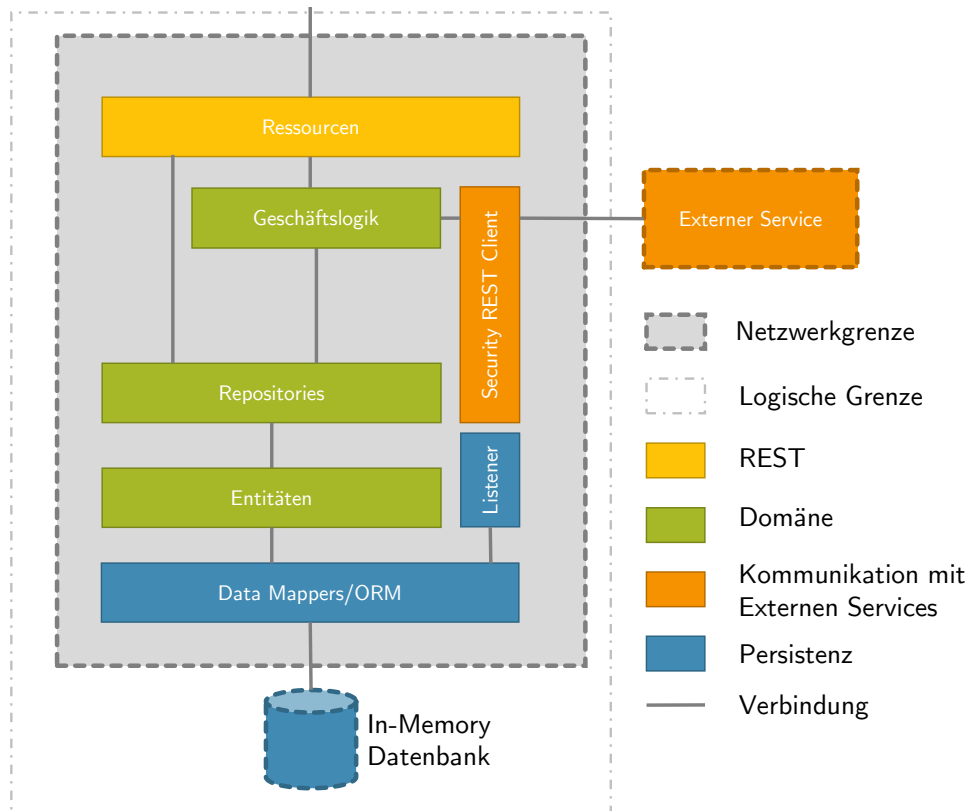


Abbildung 9: Aufbau eines Service innerhalb der it@M-Architektur

## 4 Software Testen

### 4.1 Bekannte Testmethoden

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

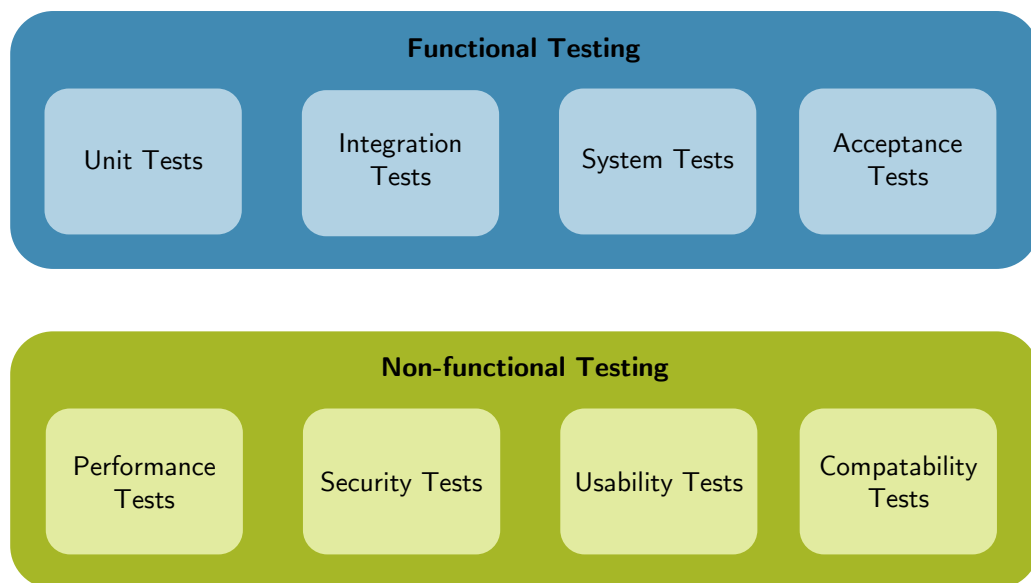


Abbildung 10: Funktionale und nicht-funktionale Testmethoden

Funktionale Tests beschäftigen sich nur mit den funktionalen Anforderungen einer Anwendung und stellen fest, ob, und wie gut diese Anforderungen erfüllt werden.[Eri12b].

5 Funktionales Testen wird häufig in vier Komponenten aufgeteilt, die auch meist in Reihenfolge abgearbeitet werden.[Inf16]

**Unit Tests** testen einzelne Module einer Software und werden meist direkt vom Entwickler 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]

15 **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]

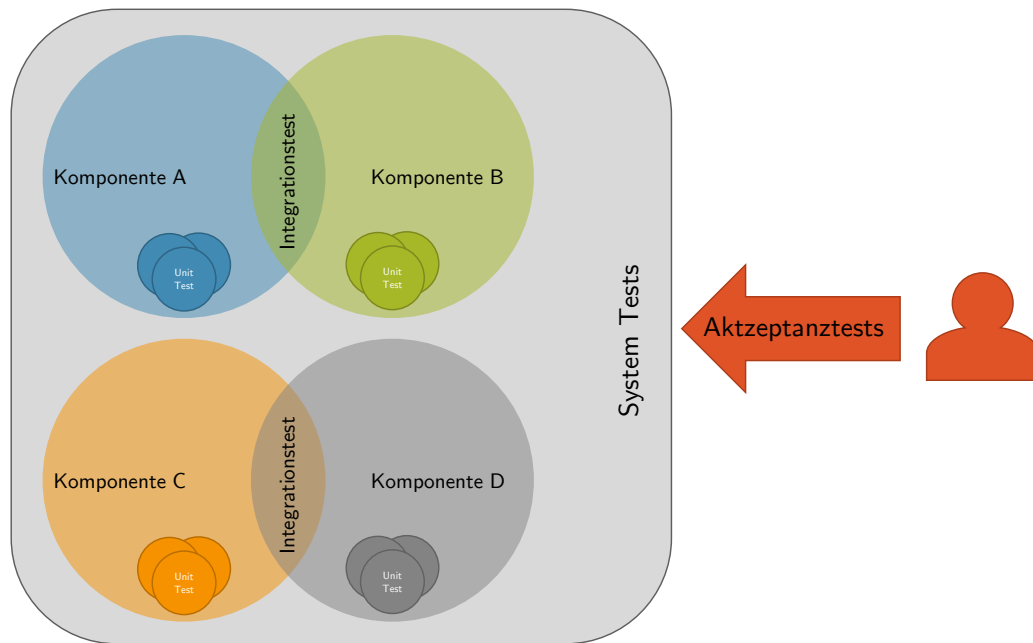


Abbildung 11: Umfang der verschiedenen Testmethoden

Nicht-funktionales Testen evaluiert die *Bereitschaft* des Systems.[Eri12b] Beispielsweise  
 5 wird durch Performance-Tests festgestellt, wie gut eine Anwendung mit vielen Anfragen  
 umgehen kann.[Inf16] Das Security Testing konzentriert sich auf das prüfen der Anwen-  
 dung in Hinblick auf Vertraulichkeit von Informationen, Integrität, Erreichbarkeit und  
 die Authentifizierung.[Inf16] Usability Tests sind eine eher objektive Testmethode mit de-  
 ren Hilfe die Benutzerfreundlichkeit einer Anwendung sichergestellt werden soll. Es wird  
 10 besonders auf die Erlernbarkeit der Anwendung, Effizienz bei der Nutzung, Zufriedenstel-  
 lung des Nutzers und Einprägsamkeit Wert gelegt. Zu guter Letzt findet im Compatibility  
 Test die Prüfung statt, ob die Anwendung auf allen geforderten Betriebssystemen, Brow-  
 sern und/oder Hardware-Plattformen lauffähig ist und alle Funktionen sich kongruent  
 verhalten.

## 15 4.2 Testen von Microservices

Durch einige spezielle Eigenschaften von Microservices ergeben sich teilweise auch spezielle  
 Testanforderungen die in der Entwicklung von Tests Beachtung finden müssen.



Tony Clemson hat im Jahr 2004 einen Vortrag .....

**4.2.0.1 Unit Testing** Auch bei Microservices beginnt das Testen eines Gesamtsystems auf der niedrigsten Schicht. Auch hier gilt es, dass Unit Tests generell auf Klassen-Level geschrieben werden.[Cle14] Im Falle des Architekturmodells von it@M beschränken sich  
 5 Unit Tests auf das Testen der Geschäftslogik, sowie der verschiedenen Listnern. Die Repositories und Entitäten sind durch das verwendete Framework nicht mit reinen Unit Tests testbar, sondern werden in den folgenden Integrationstests durch Mocking getestet.

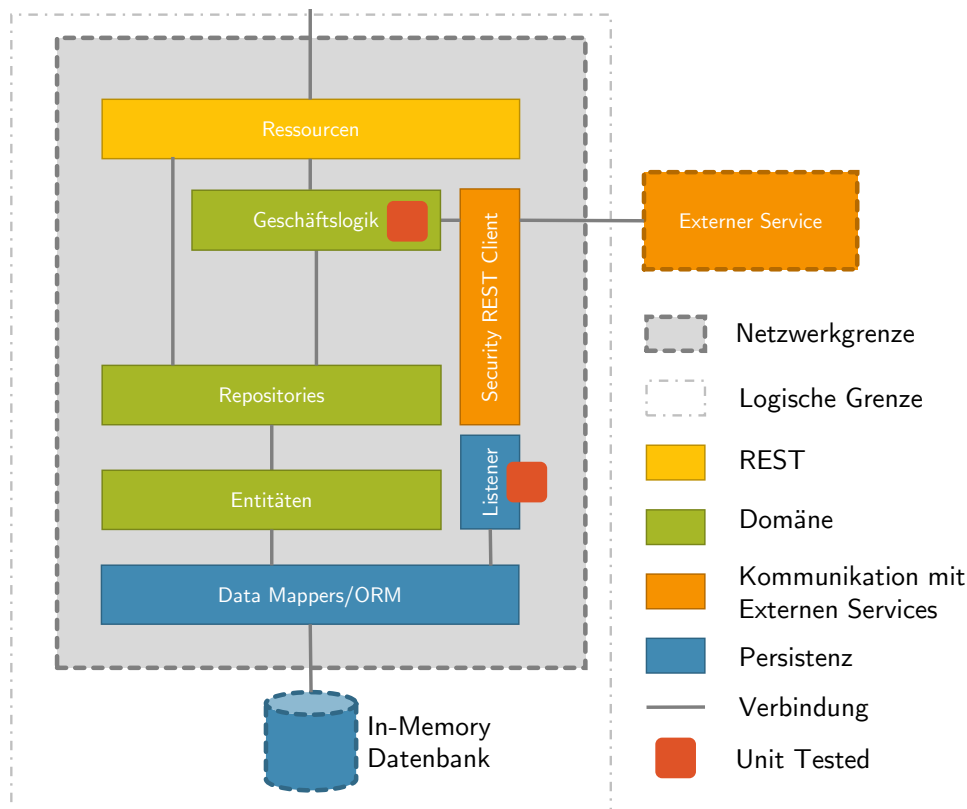


Abbildung 12: Unit Testing Scope [Cle14]

**4.2.0.2 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  
 15 hier wird also nicht vom Regelfall aus Kapitel 4.1 abgewichen. Auch wird in diesem Stadium das Zusammenspiel des Data Mappers und der angebundenen Datenbank, sowie die daraus resultierenden Ressourcen auf Korrektheit geprüft. 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]

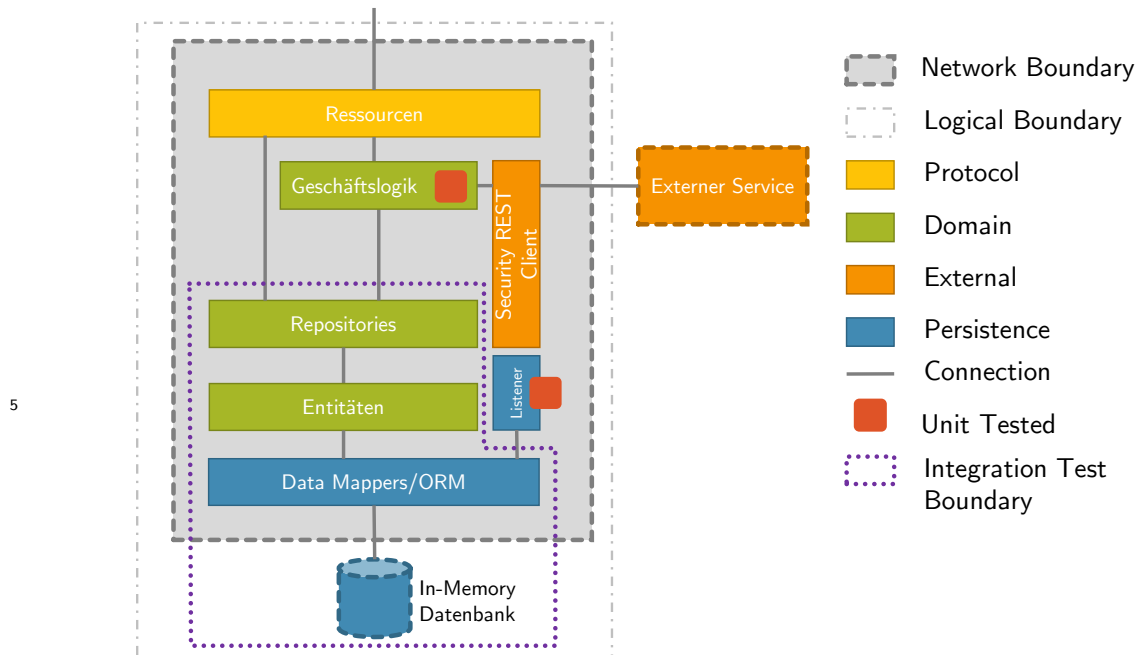


Abbildung 13: Integration Testing Scope [Cle14]

**4.2.0.3 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 Monolithen unterscheidet. Nur, weil der getestete Service alle Anforderungen erfüllt, gilt dies noch nicht für die Services zu denen Abhängigkeiten bestehen. Das Problem an dieser Stelle: End-to-End-Tests, also Tests bei denen die gesamte Domäne gestartet und getestet wird sind sehr aufwändig und bieten nur sehr lange Feedback-Zyklen, im Gegensatz zu automatisierten Testmethoden.[Cle14]

Component testing tritt an dieser Stelle ein. Es eignet sich besonders gut für Microservice Architekturen, da die sich diese leicht in die getrennt zu testenden Komponenten aufteilen lassen: Jeder Service entspricht einer Komponente.[Cle14]

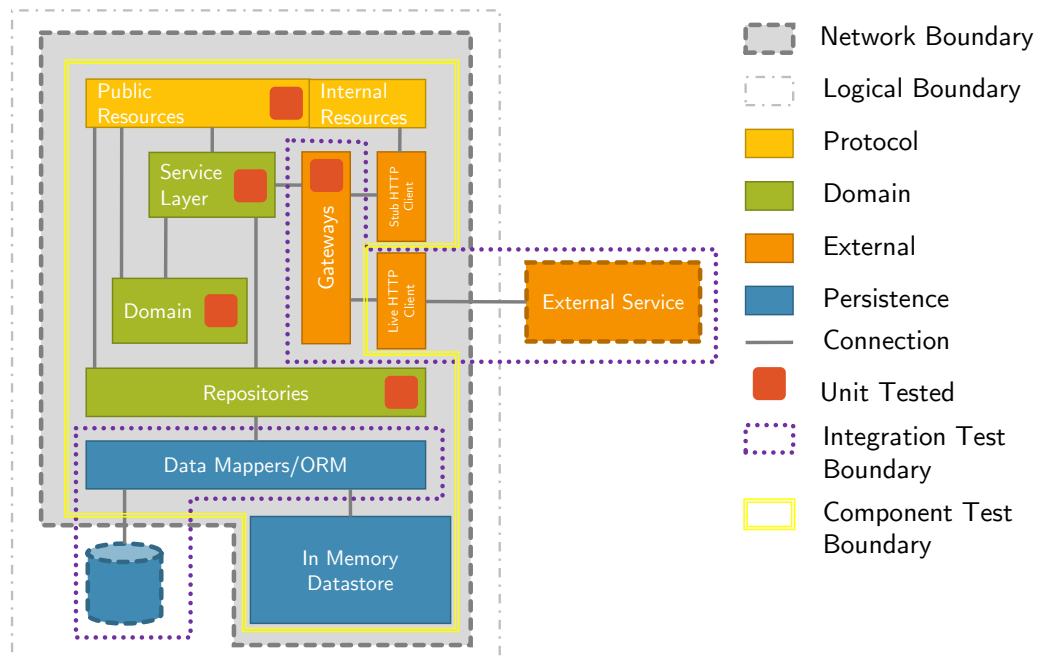


Abbildung 14: Component Testing Scope [Cle14]

#### 4.2.0.4 Contract Testing hier kommt noch text hin

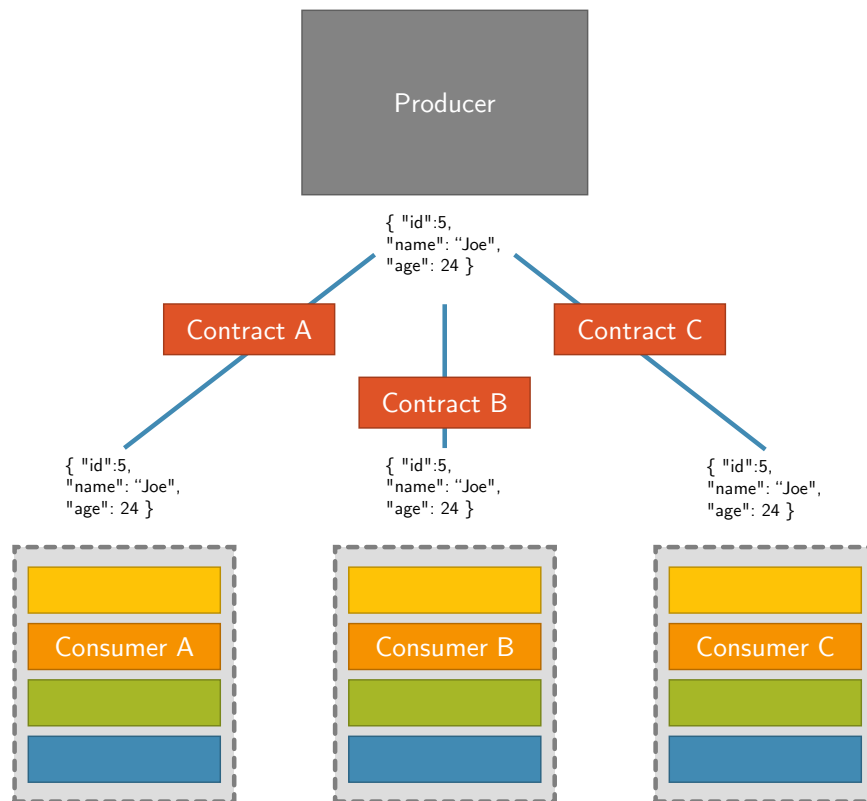


Abbildung 15: Contract Testing [Cle14]

#### 4.2.0.5 End-To-End Testing

hier kommt noch text hin

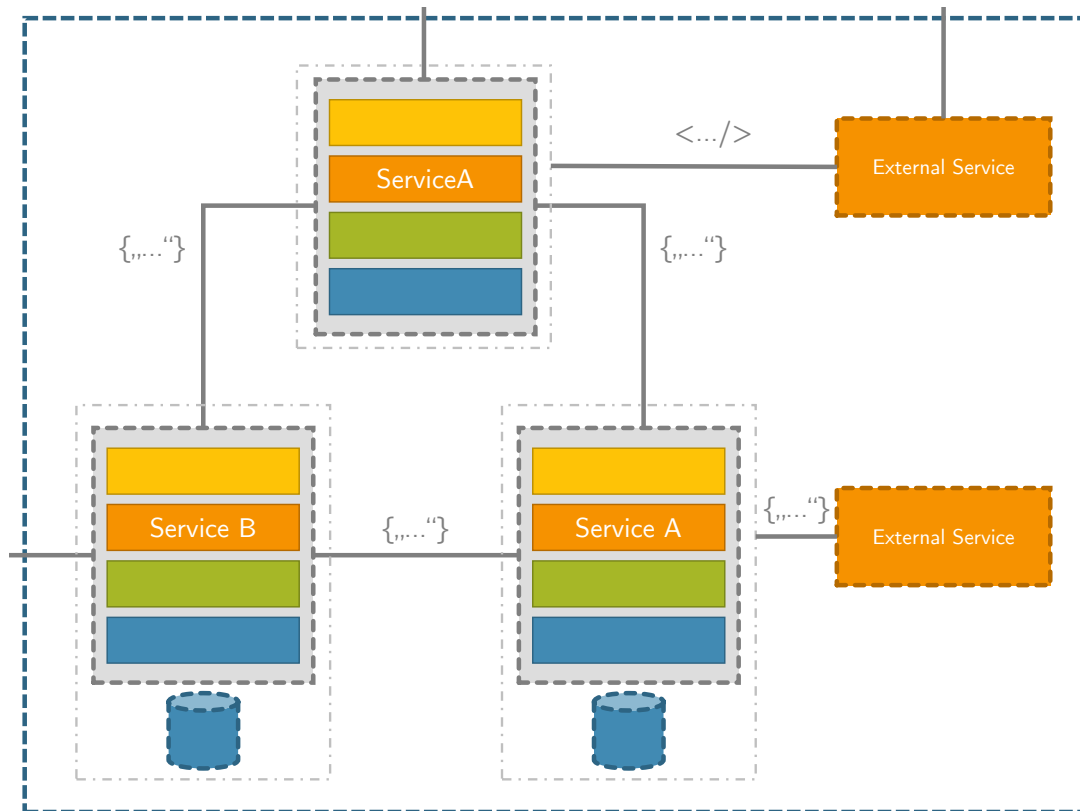


Abbildung 16: End-To-End Testing Scope [Cle14]

#### 4.2.1 Sinnvolle Teststrategien für den generativen Ansatz

#### 4.2.2 Frameworks zum Umsetzen von Test-Strategien

### 5 Anforderungen an generierte Tests

#### 5.1 Benötigte Daten

#### 5.2 Notwendige Änderungen/Erweiterungen von Barrakuda

## 6 Implementierung in Barrakuda

### 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 HATEOAS<sup>3</sup> genutzt.

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

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

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

Clients können mit der Domäne über ein API-Gateway kommunizieren (Netflix Zuul). Dieses öffnet nach außen eine einzige Schnittstelle, über die sowohl eine graphische Nutzeroberfläche aufgerufen, als auch die einzelnen Services kontaktiert werden können.  
15 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  
20 Container-Frameworks zu ermöglichen.

### 6.1.2 Beispiel-Anwendung

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

---

```

25 1 domain kongo package edu.hm.ba version 1.0;
2
3 serviceModel shoppingcart package edu.hm.ba.kongo.shop version 1.0 {
4
30 5     customNumberType intMin1 minValue=1;
6
7     entity Cart {
8         items manyToMany CartItem;
9         userID warehouse.OID "123";
35 10        totalPrice warehouse.float "15.5";

```

---

<sup>3</sup><http://projects.spring.io/spring-hateoas/>

```

11     }
12
13     valueObject CartItem {
14         product warehouse.OID mainFeature "123";
5 15         quantity intMin1 mainFeature "1";
16     }
17
18     businessAction addToCart {
19         purpose "Add a product from the warehouse to the users current
10         shopping cart";
20         given productID warehouse.OID;
21         given quantity intMin1;
22     }
23
15 24 }
25
26 serviceModel ordering package edu.hm.ba.kongo.shop version 1.0{
27
28     customTimeType dateInPast inThePast;
20 29
30     entity order {
31         cart warehouse.OID "123";
32         orderedOn dateInPast "10.10.2010";
33     }
25 34
35     businessAction orderCart {
36         purpose "Receives a shopping cart to create a new order which can
           then be payed";
37         given cartID warehouse.OID;
30 38     }
39
40     businessAction sendInvoice{
41         purpose "Sends the value of the costs of the ordered procuts to
           an invoicing system";
35 42         given orderID warehouse.OID;
43     }
44
45     businessAction cancelOrder{
46         purpose "Deletes an order and the associated shopping cart with
40         it 's contents";
47         given orderID warehouse.OID;
48     }
49
50 }
45 51

```

```
52 serviceModel warehouse package edu.hm.ba.kongo.shop version 1.0{
53
54     customTextType OID maxLength=24;
55     customTextType stringMin1 minLength=1;
5 56     customTextType longText type=long;
57     customNumberType float pointNumber minValue=0;
58     customNumberType int minValue=0;
59
60     entity Product {
10 61         name stringMin1 searchable mainFeature "Bottle";
62         description longText optional "Great for storing water";
63         price float mainFeature "10.5";
64         quantity int mainFeature "5";
65     }
15 66
67 }
```

---

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

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

Der mit generierte Authentifizierungsservice dient als Kundenverwaltung.

Wichtig ist, dass im Referenz-System keinerlei Logik zu den modellierten Geschäftsanwendungen implementiert wird, sondern lediglich die im Abschnitt 4.2 angesprochenen Testmethoden.



### 6.1.3 Komponenten und Aufbau

### 6.1.4 Implementierung des Systems

### 6.1.5 Implementierung der Tests

## 6.2 Übernahme der Referenz-Implementierung in

### 5 Barrakuda-Templates

## 7 Fazit

## 8 Quellenverzeichnis

- [Cha05] CHARETTE, Robert N.: Why Software Fails. (2005). <http://spectrum.ieee.org/computing/software/why-software-fails>
- 10 [Cle14] CLEMSON, Toby: Testin 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
- 15 [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
- 20 [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
- 25 [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
- 30

- [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](http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm). – Zuletzt Abgerufen am 13.01.2017
- 5 [Fow15] FOWLER, Martin: Microservices Trade-Offs. (2015). <https://martinfowler.com/articles/microservice-trade-offs.html>. – Zuletzt Abgerufen am 24.01.2017
- [Hag08] HAGEN, Stefan: Das "Triple Constraint" im Projektmanagement. (2008). [http://pm-blog.com/2008/08/04/triple\\_constraint\\_magisches\\_dreieck/](http://pm-blog.com/2008/08/04/triple_constraint_magisches_dreieck/). – Zuletzt Abgerufen am 10.01.2017
- 10 [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
- 15 [Inf16] INFLECTRA: Testing Methodologies. (2016). <https://www.inflectra.com/Ideas/Topic/Testing-Methodologies.aspx>. – Zuletzt Abgerufen am 26.01.2017
- 20 [Kur16] KURZ, Martin: *Verbesserung des Softwareentwicklungsprozesses der Landeshauptstadt Muenchen durch modellgetriebene Softwareentwicklung*, Hochschule München, Diplomarbeit, 2016
- [New15] NEWMAN, Sam: *Building Microservices*. O'Reilly, 2015. – ISBN 9781491950357
- [Pan99] PAN, Jiantao: Software Testing. (1999). [https://users.ece.cmu.edu/~koopman/des\\_s99/sw\\_testing/](https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/). – Zuletzt Abgerufen am 26.01.2017
- 25 [Ric14] RICHARDSON, Chris: Pattern: Monolithic Architecture. (2014). <http://microservices.io/patterns/monolithic.html>. – Zuletzt Abgerufen am 16.01.2017
- [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
- 30 [SQS] SQS: Detect errors early on, reduce costs and increase quality. <https://www.sqs.com/en/academy/download/fact-sheet-EED-en.pdf>

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

## **Anhang**

### **A Code-Fragmente**

Viel Beispiel-Code