Fakultät für Informatik und Mathematik 07

Bacheloararbeit

über das Thema

Generative Testerstellung für Microservice-Architekturenit

Autor: Fabian Wilms
holtkoet@hm.edu

Prüfer: Prof. Dr. Ulrike Hammerschall

Abgabedatum: xx.xx.2017

I Kurzfassung

kurzfassung

Abstract

Das ganze auf Englisch.

II Inhaltsverzeichnis

I Kurzfassung **I**

II Inhaltsverzeichnis **II**

III Abbildungsverzeichnis **III**

IV Tabellenverzeichnis **III**

V Listing-Verzeichnis **III**

VI Abkürzungsverzeichnis **III**

1 Einführung und Motivation **1**

2 Software Testen **3**
2.1 Bekannte Testmethoden . . . 3
2.2 Testen von Microservices . . . 4
2.2.1 Sinnvolle Teststrategien für den generativen Ansatz . . . 10
2.2.2 Frameworks zum Umsetzen von Test-Strategien . . . 10

3 Architektur **10**
3.1 Microservice Architektur und Unterschiede zur monolithischen Architektur 10
3.2 Vorgegebene Architektur der LHM . . . 12
3.2.1 Authentifizierungsservice . . . 12
3.2.2 Discoveryservice . . . 12
3.2.3 Configurationservice . . . 12
3.2.4 Service . . . 12
3.2.5 Webservice . . . 12
3.3 Unterschiede zwischen Microservices und Monolith-Architekturen . . . 12
3.3.1 Auswirkungen . . . 12

4 Anforderungen an generierte Tests **12**
4.1 Benötigte Daten . . . 12
4.2 Notwendige Änderungen/Erweiterungen von Barrakuda . . . 12

5 Implementierung in Barrakuda **12**
5.1 Referenz-System . . . 12
5.1.1 Komponenten und Aufbau . . . 12
5.1.2 Implementierung des Systems . . . 12
5.1.3 Implementierung der Tests . . . 12
5.2 Übernahme der Referenz-Implementierung in Barrakuda-Templates . . . 12

6 Quellenverzeichnis **12**

Anhang **I**

A Code-Fragmente **I**

III Abbildungsverzeichnis

Abb. 1 SQS Report Costs of Defect Correction 1
Abb. 2 Testmethoden 3
Abb. 3 Funktionale Testmethoden 4
Abb. 4 Unit Testing Scope 5
Abb. 5 Integration Testing Scope 6
Abb. 6 Component Testing Scope 7
Abb. 7 Contract Testing 8
Abb. 8 End-To-End Testing Scope 9
Abb. 9 Monolithische Architektur 10
Abb. 10 Microservice Architektur 11

IV Tabellenverzeichnis

V Listing-Verzeichnis

VI Abkürzungsverzeichnis

1 Einführung und Motivation

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

Mit steigender Komplexität und höherer Nachfrage am Markt, sowie engen Zeitplänen für Projekte wird leider häufig aus Zeit- und Kostengründen auf Qualität nur geringfügig Rücksicht genommen. Zunächst verursacht eine gute Software-Qualität nämlich nur Mehrkosten. Personelle wie zeitliche. Die langfristige Sinnhaftigkeit bleibt dabei außen vor, aus der Vergangenheit wird nur selten gelernt.

Ein Bericht der Kölner Beratungsfirma SQS zeigt anhand von gesammelten Zahlen aus Beratungsaufträgen welche immensen Kosten durch unentdeckte Fehler entstehen. Hier wird besonders deutlich wie viel es für ein Projekt bedeutet, frühzeitige Qualitätssicherung durchzusetzen. Und dazu zählt auch das Testen von Software.

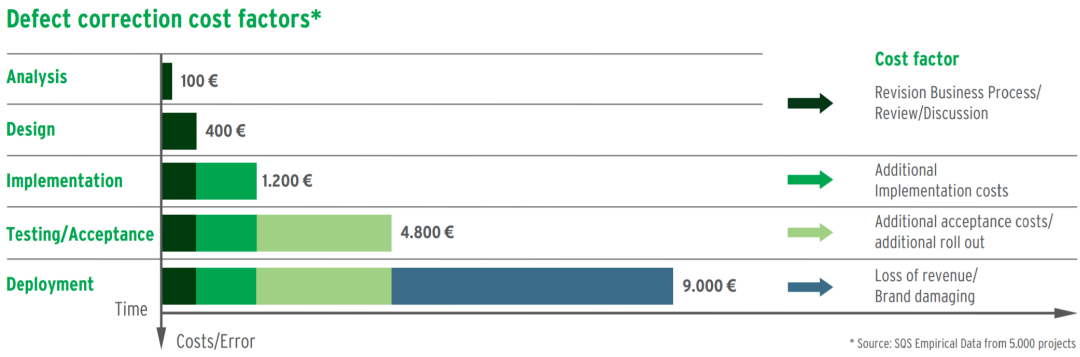

Abbildung 1: SQS Report Costs of Defect Correction [SQS]

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€ 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*.

> *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.[Cha05]*

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. Und Testen kostet neben Geld auch Zeit. Es entsteht in dieser Zeit aber kein Fortschritt an der Funktionalität der Software. Dies ist auch der Grund, weshalb das Testen bei Entwicklern nicht an oberster Stelle der liebsten Aufgaben steht. Es kostet Zeit ohne erkennbaren Fortschritt zu erreichen.

Bei it@M, dem Eigenbetrieb der Landeshauptstadt München, ist eine hohe Testabdeckung daher Teil der Definition of Done [citation needed] von Softwareprojekten. Im Kommunalen Umfeld sind die zeitlichen Restriktionen noch einmal stärker zu gewichten als in der freien Wirtschaft. Viele Projekte werden aufgrund von anstehenden Gesetzesänderungen ins Leben gerufen und müssen mit Inkrafttreten der neuen Regelungen in Produktion gehen. it@M ist somit ständig auf der Suche nach Lösungen, die den Entwicklungs- und Testprozess beschleunigen, um die geringe Zeit möglichst effizient nutzen zu können.

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

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

Zunächst werden bekannte Methoden zum testen von Software analysiert und in Hinblick der Umsetzungsmöglichkeit im generativen Ansatz geprüft. Anschließend werden alternative Methoden, die besonders im Bereich der Microservices anzutreffen sind, ebenfalls untersucht.

Im nächsten Schritt wird die von it@M Vorgegebene Architektur genauer erläutert und die zu testenden Komponenten identifiziert.

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

Schließlich sollen Anforderungen, die an die zu generierenden Tests gestellt werden festgehalten werden, die im letzten Abschnitt, der Implementierung, Beachtung finden werden.

2 Software Testen

2.1 Bekannte Testmethoden

5 //TODO

unterschied funktional, nichtfunktional...

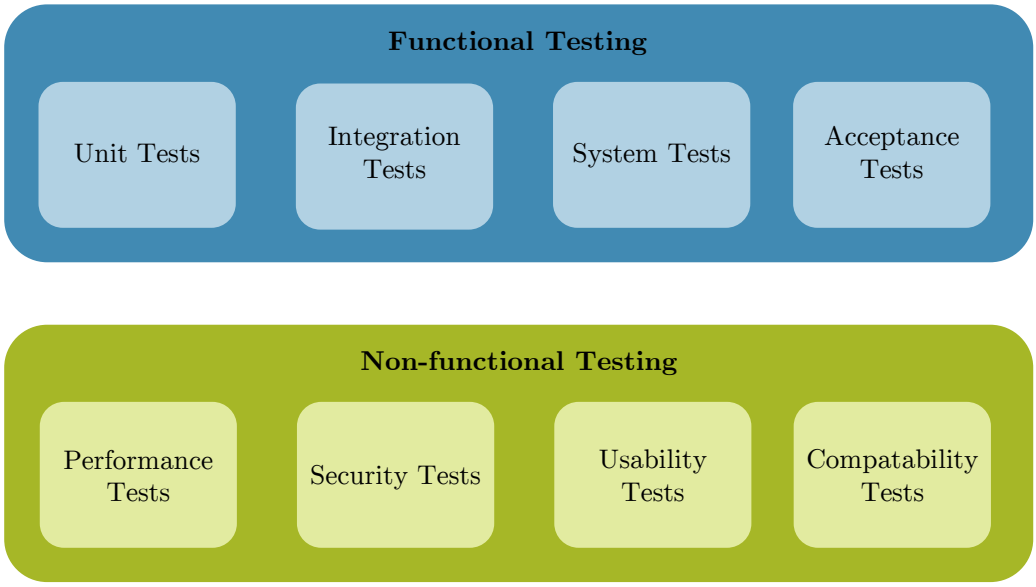

Abbildung 2: Testmethoden[?]

detailiert funktionale tests

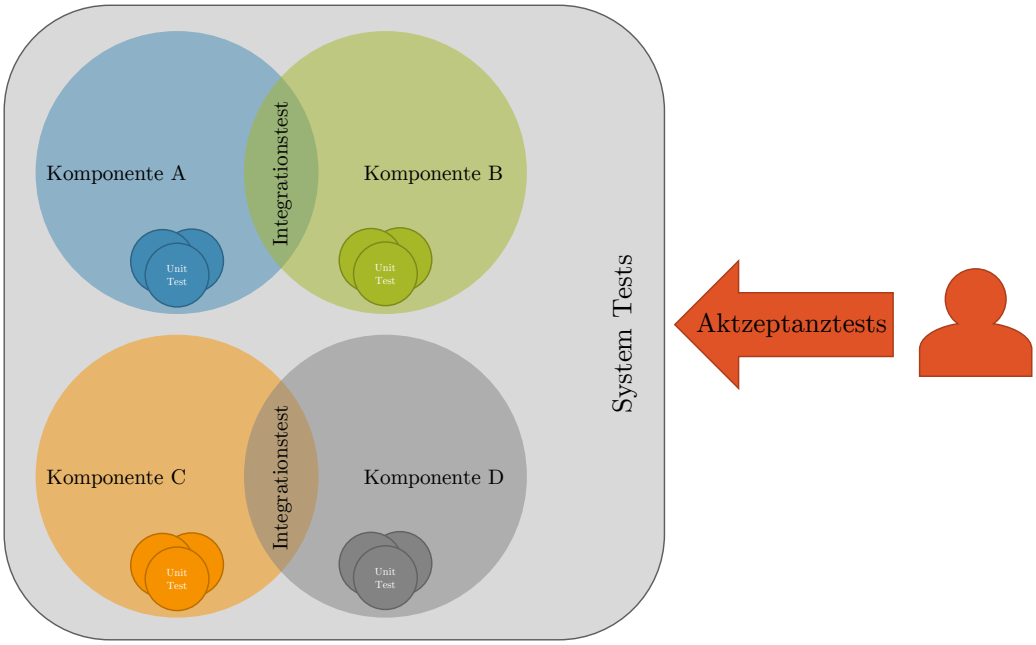

Abbildung 3: Funktionale Testmethoden

2.2 Testen von Microservices

Unit Tests hier steht text

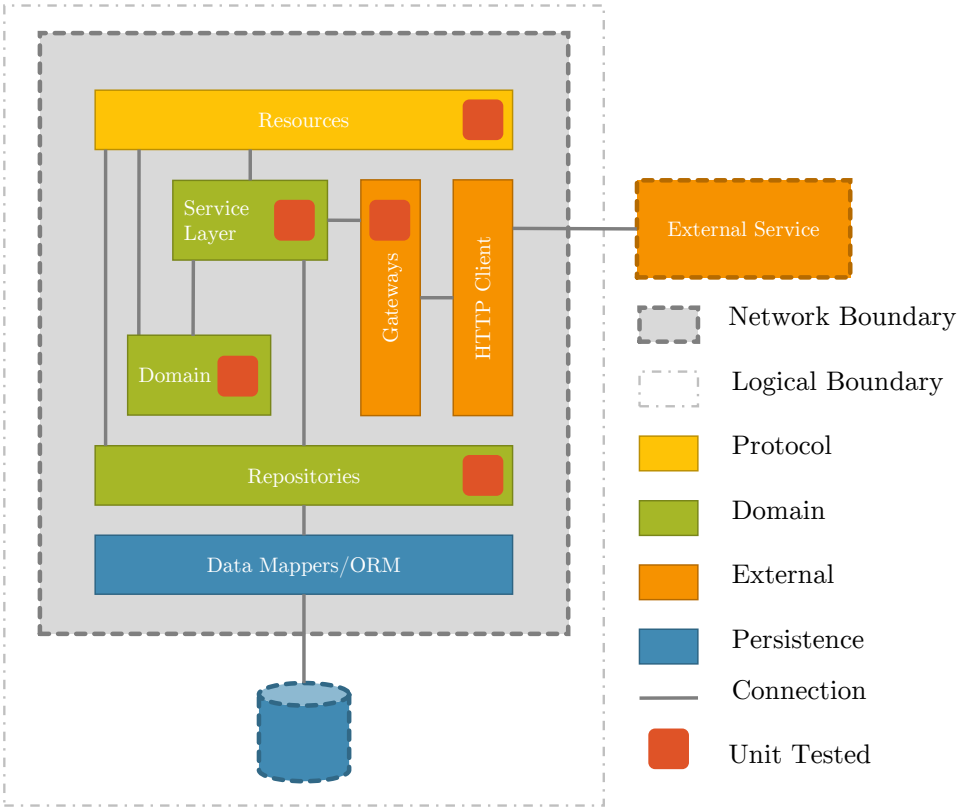

Abbildung 4: Unit Testing Scope [Cle14]

Integrationstests text

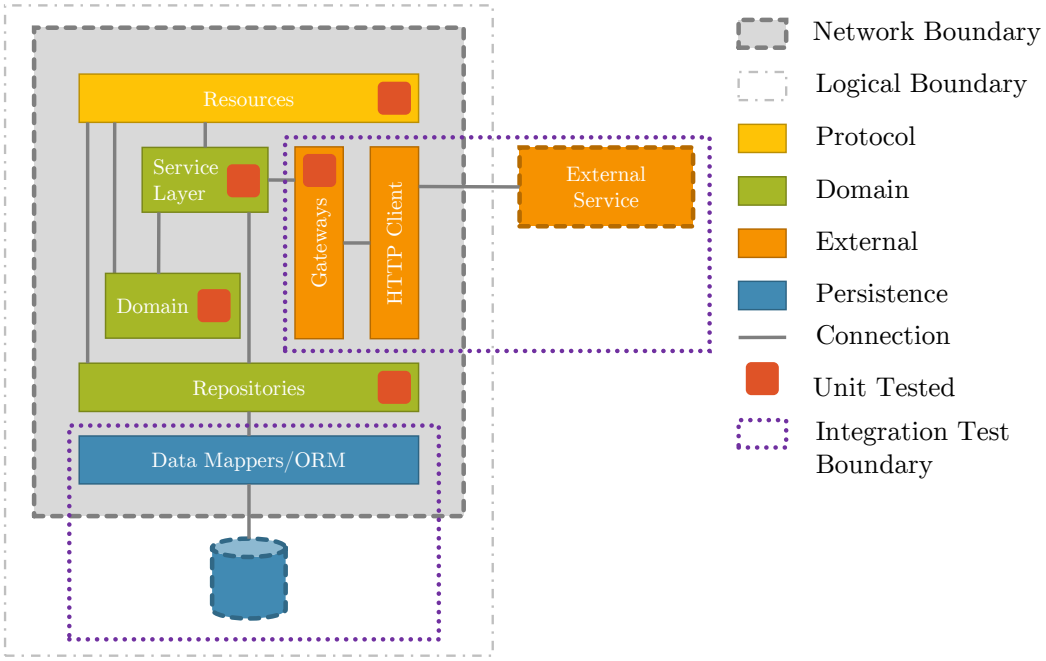

Abbildung 5: Integration Testing Scope [Cle14]

Komponententests text

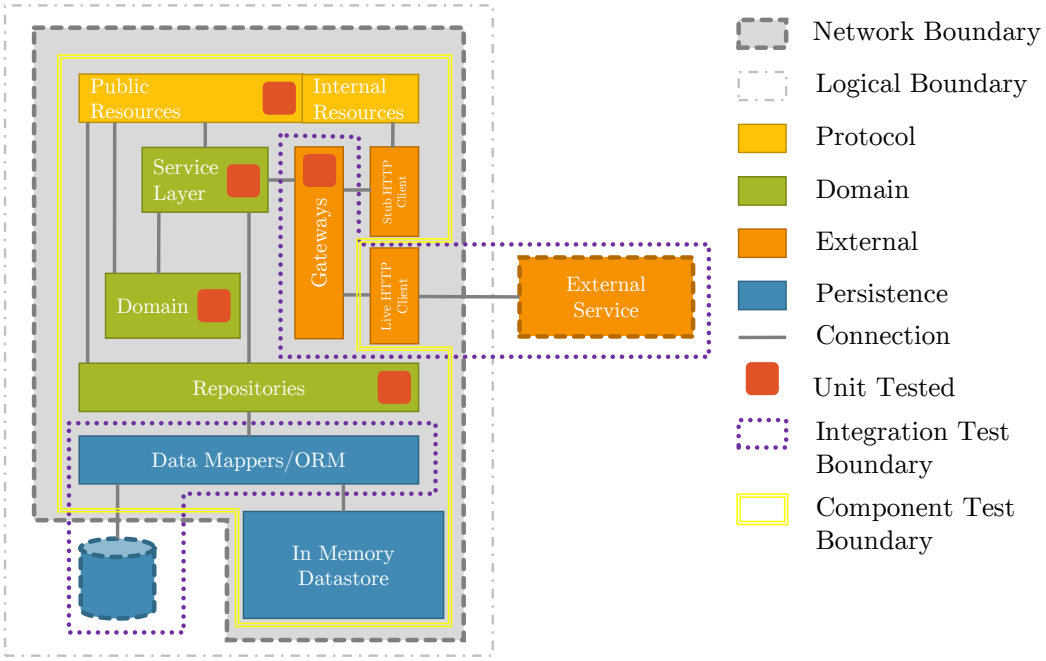

Abbildung 6: Component Testing Scope [Cle14]

Contract Testing text

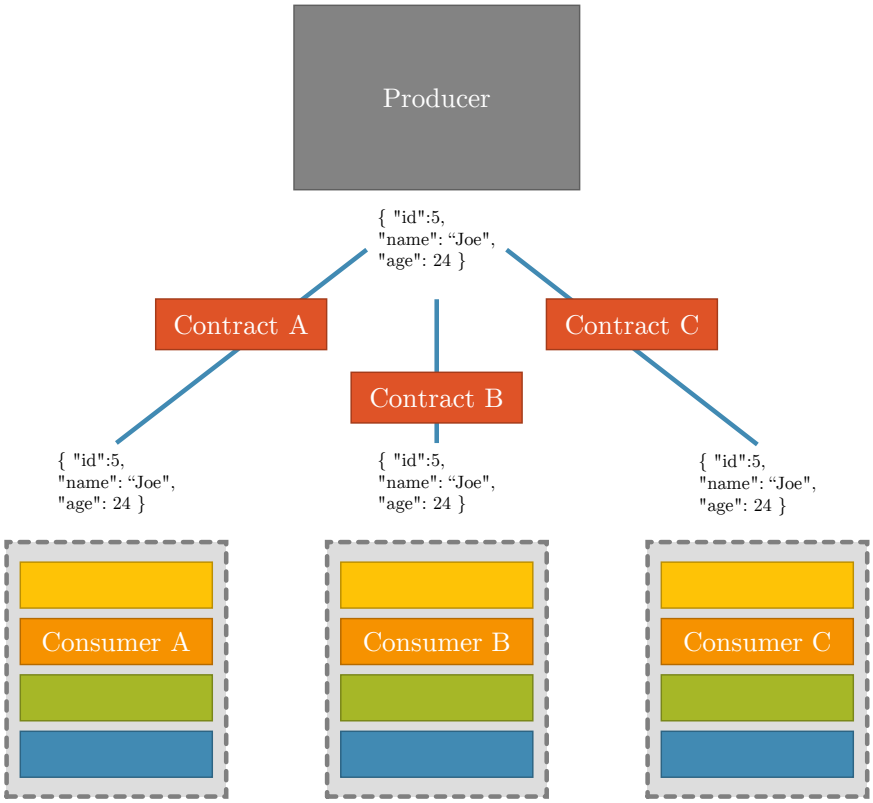

Abbildung 7: Contract Testing [Cle14]

End-To-End Testing text

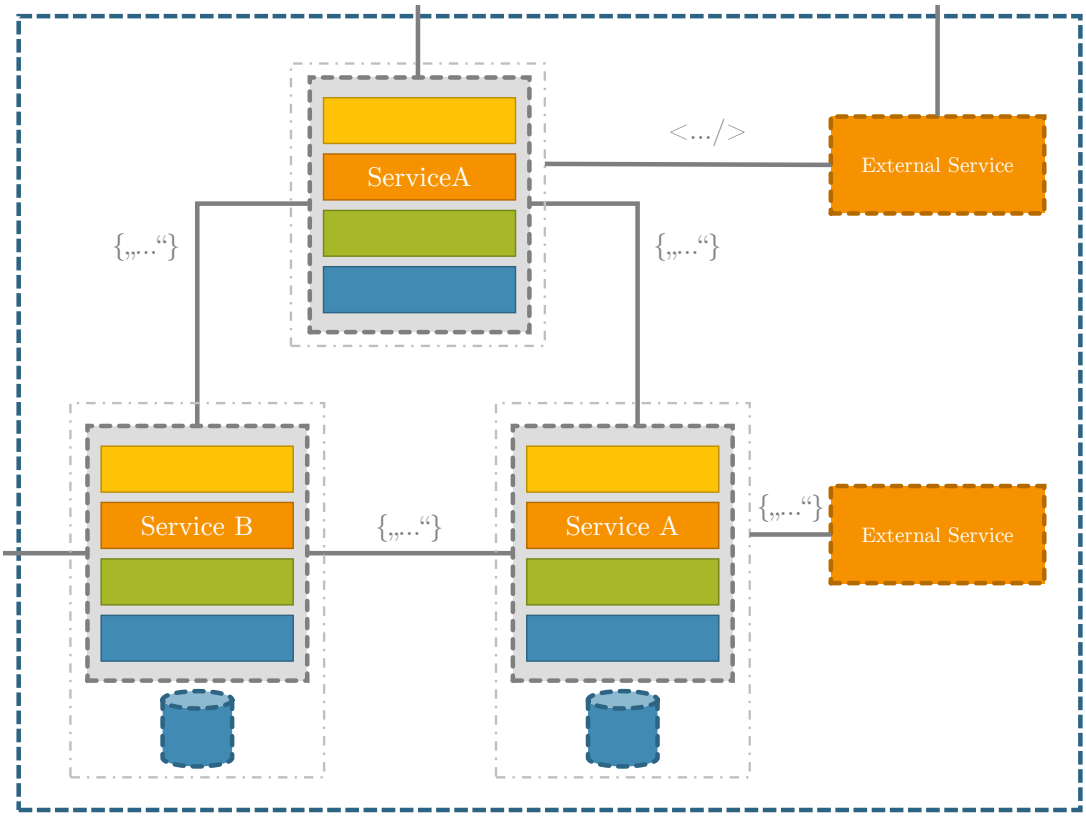

Abbildung 8: End-To-End Testing Scope [Cle14]

2.2.1 Sinnvolle Teststrategien für den generativen Ansatz

2.2.2 Frameworks zum Umsetzen von Test-Strategien

3 Architektur

3.1 Microservice Architektur und Unterschiede zur monolithischen Architektur

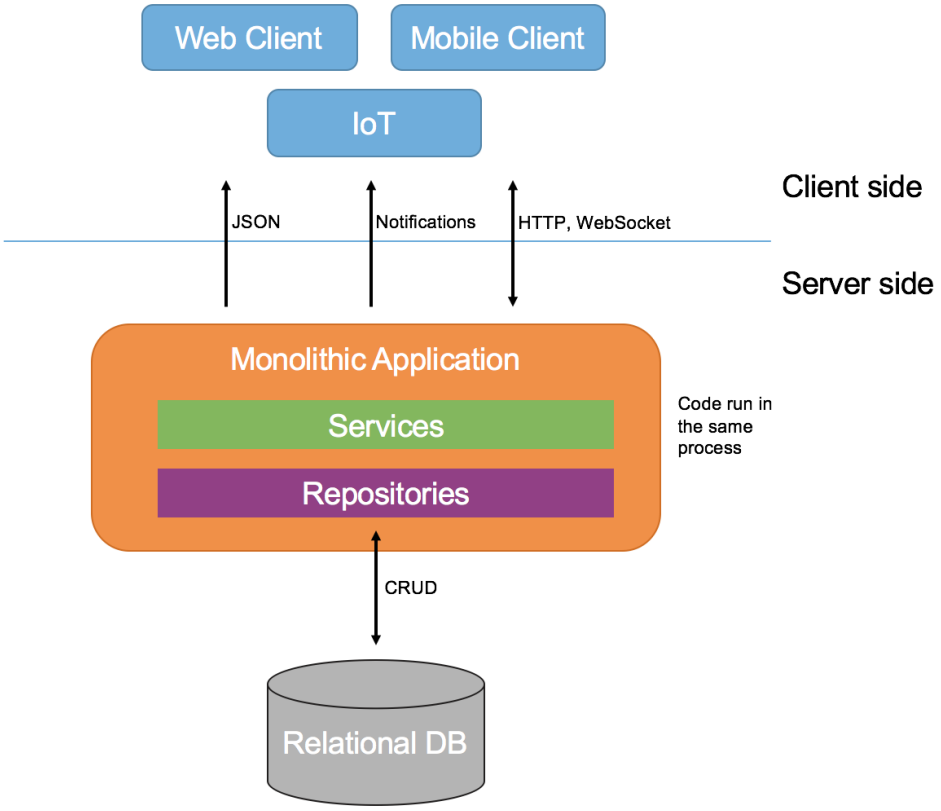

Abbildung 9: Monolithische Architektur [DIN15]

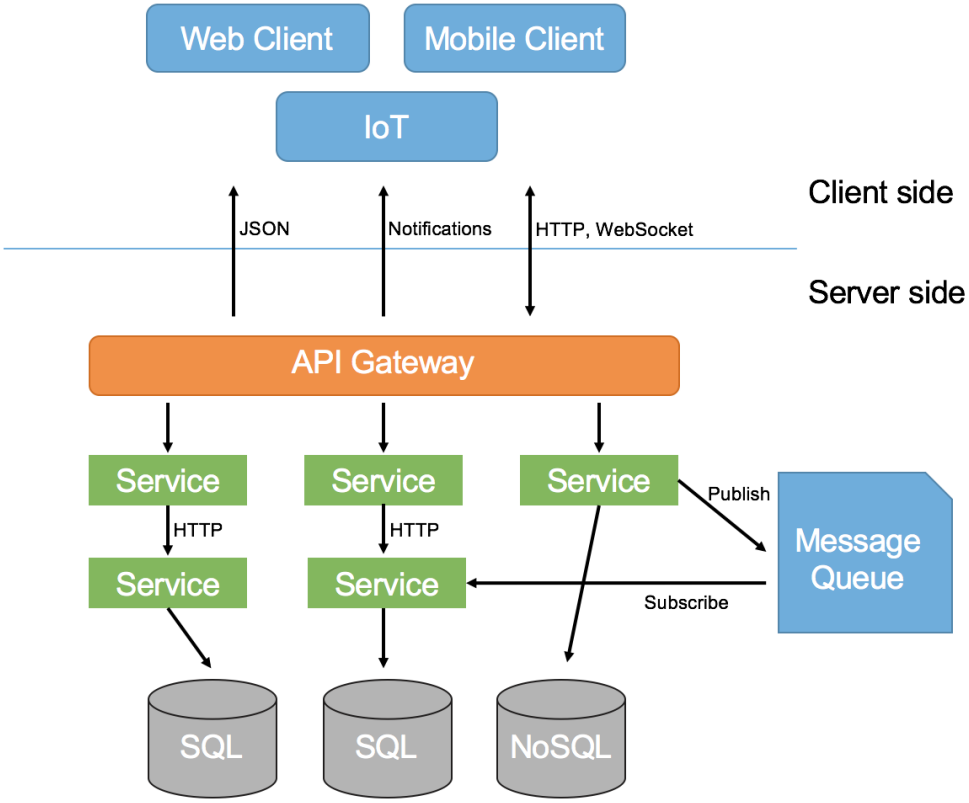

Abbildung 10: Microservice Architektur [DIN15]

3.2 Vorgegebene Architektur der LHM

3.2.1 Authentifizierungsservice

3.2.2 Discoveryservice

3.2.3 Configurationservice

3.2.4 Service

3.2.5 Webservice

3.3 Unterschiede zwischen Microservices und Monolith-Architekturen

3.3.1 Auswirkungen

4 Anforderungen an generierte Tests

4.1 Benötigte Daten

4.2 Notwendige Änderungen/Erweiterungen von Barrakuda

5 Implementierung in Barrakuda

5.1 Referenz-System

5.1.1 Komponenten und Aufbau

5.1.2 Implementierung des Systems

5.1.3 Implementierung der Tests

5.2 Übernahme der Referenz-Implementierung in Barrakuda-Templates

6 Quellenverzeichnis

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

[Cle14] Clemson, Toby: Testin Strategies in a Microservice Architecture. (2014). http://martinfowler.com/articles/microservice-testing/. – Abgerufen am 01.11.2016

[DIN15] DINH, KHOA: An Overview of Microservices Architecture. (2015). http://khoadinh.github.io/2015/05/01/microservices-architecture-overview.html. – Abgerufen am 18.11.2016

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

Anhang

A Code-Fragmente

Viel Beispiel-Code