Fakultät für Informatik und Mathematik 07

Bacheloararbeit

über das Thema

Generative Testerstellung für Microservice-Architekturenit

Autor: Fabian Wilms
holtkoet@hm.edu

Prüfer: Prof. Dr. Ulrike Hammerschall

Abgabedatum: xx.xx.2017

I Kurzfassung

kurzfassung

Abstract

Das ganze auf Englisch.

II Inhaltsverzeichnis

III Abbildungsverzeichnis

IV Tabellenverzeichnis

V Listing-Verzeichnis

VI Abkürzungsverzeichnis

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.

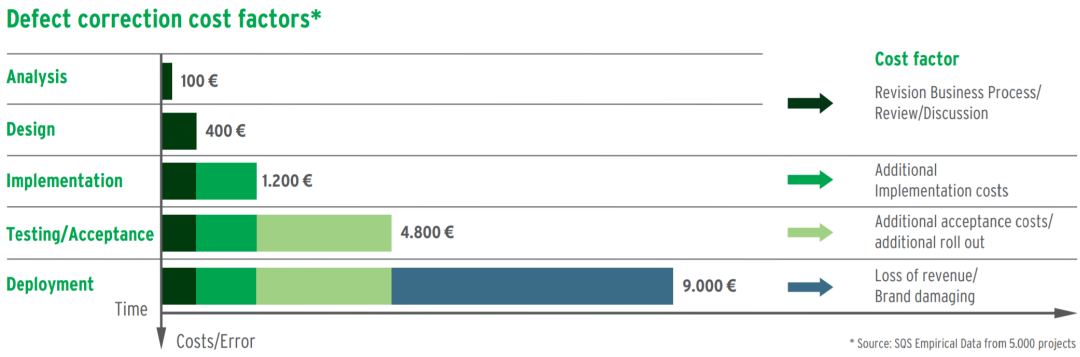

Abbildung 1: SQS Report Costs of Defect Correction [SQS]

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€
citesqsdefect 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. 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[Kur16] geplant und entwickelt. Die Model-driven Software Development Lösung Barrakuda [1]. Diese bietet den Entwicklern von it@M die Möglichkeit anhand von einer vorgegebenen Domänen-spezifischen Sprache Microservice-Architekturen zu modellieren und diese zu generieren.

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

Zunächst wird die Microservice Architektur im generellen und anschließend im speziellen anhand der von it@M Vorgegebene Architektur genauer erläutert und die zu testenden Komponenten identifiziert.

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

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

untersucht.

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

2 Architektur

Sam Newman stellt in seinem Buch *Building Microservices* einige Vorteile dar, die Microservices gegenüber Monolithischen Architekturen bieten[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 mit Barrakuda eine Microservice-Architektur für it@M generiert wird. Sie behandelt einen Großteil der Probleme, die bei bestehenden Systemen der Landeshauptstadt aufgetreten sind.

2.1 Microservice Architektur und Unterschiede zur monolithischen Architektur

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

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

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

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

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

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

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

2.1.1 Kommunikation zwischen Services

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

2.1.1.1 Geteilte Datenbanken werden dafür gerne verwendet. [New15, S.85]. Dies birgt allerdings viele Probleme. Jede noch so kleine Änderung an der "Logik"dieser Datenbank, oder an der internen Struktur der Daten muss mit viel Bedacht durchgeführt werdenm, da jede abhängige Komponente sonst nicht mehr funktionieren könnte.

Des weiteren ist die technologische Einschränkung ein großer Nachteil. Wenn auch es 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 Kommunikationsmöglichkeit. Ein Vorteil von RPC ist, dass es sehr einfach und schnell möglich ist Methoden der Services für Clients und andere Services Verfügbar zu machen. Über die Kommunikation muss man sich nahezu keine Gedanken machen[New15, S.91].

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

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

2.1.1.3 REpresentational State Transfer (REST) ist eine der meistgenutzten Architekturstilen für APIs [DuV10]. 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. machen es sehr einfach 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 Hypermedia As the Engine of Application State (HATEOAS). HATEOAS ist ein Teil der REST-Spezifizierung und beschreibt ein Konzept, nach dem, einfach gesagt, Informationen Links zu anderen Informationen enthalten. 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.

```
{
  "name": "Hans Meier",
  "mail": "hansmei@mail.de",
  "links": [{
    "rel": "orders",
    "href": "https://mycompany.com/orders/12341232/"
  }]
}
```

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 örders"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 den Weg der Kommunikation gefallen ist muss die Art der Kommunikation festgelegt werden. JSON und XML sind dabei die größten Namen. In den vergangenen Jahren ist dabei die Verwendung von XML im Gegensatz zu JSON in APIs zurückgegangen[DuV13]

2.2 Vorgegebene Architektur der LHM

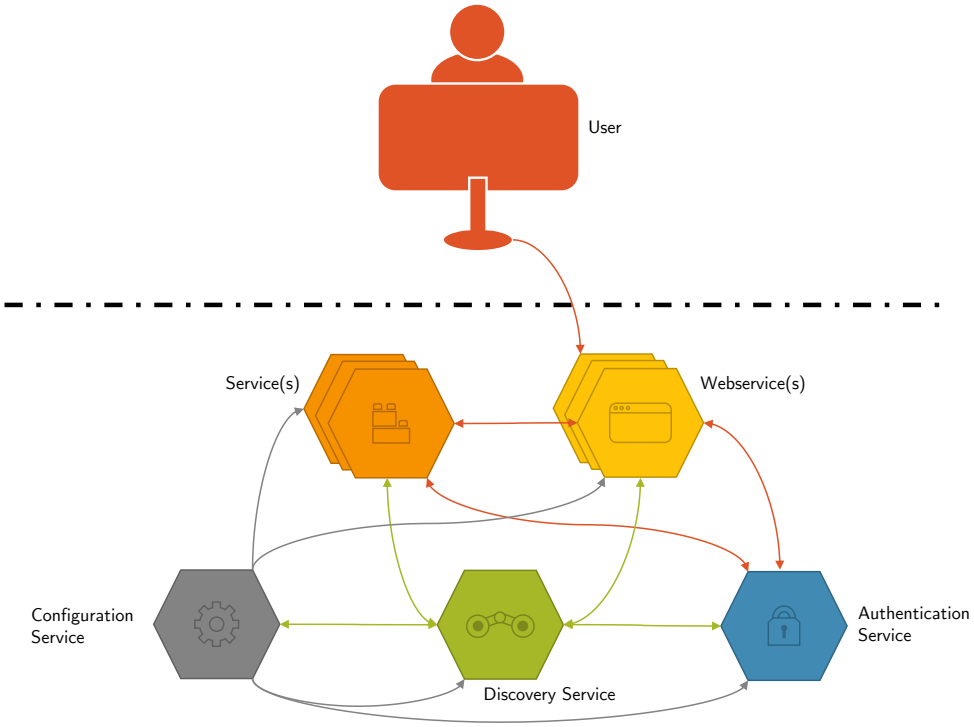

Abbildung 2: Architekturvorgabe it@M

2.2.1 Authentifizierungsservice

2.2.2 Discoveryservice

2.2.3 Configurationservice

2.2.4 Service

2.2.5 Webservice

2.3 Unterschiede zwischen Microservices und Monolith-Architekturen

2.3.1 Auswirkungen

3 Software Testen

3.1 Bekannte Testmethoden

performance, security, usability, compatability...

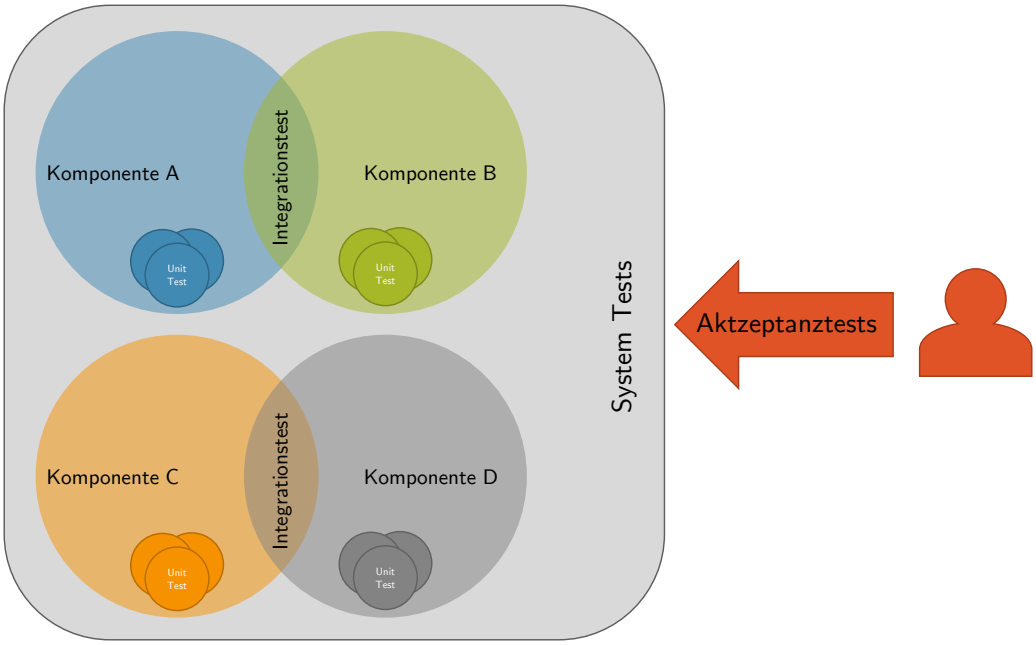

Abbildung 3: Bekannte Testmethoden

3.2 Testen von Microservices

3.2.0.1 Unit Tests

hier steht text

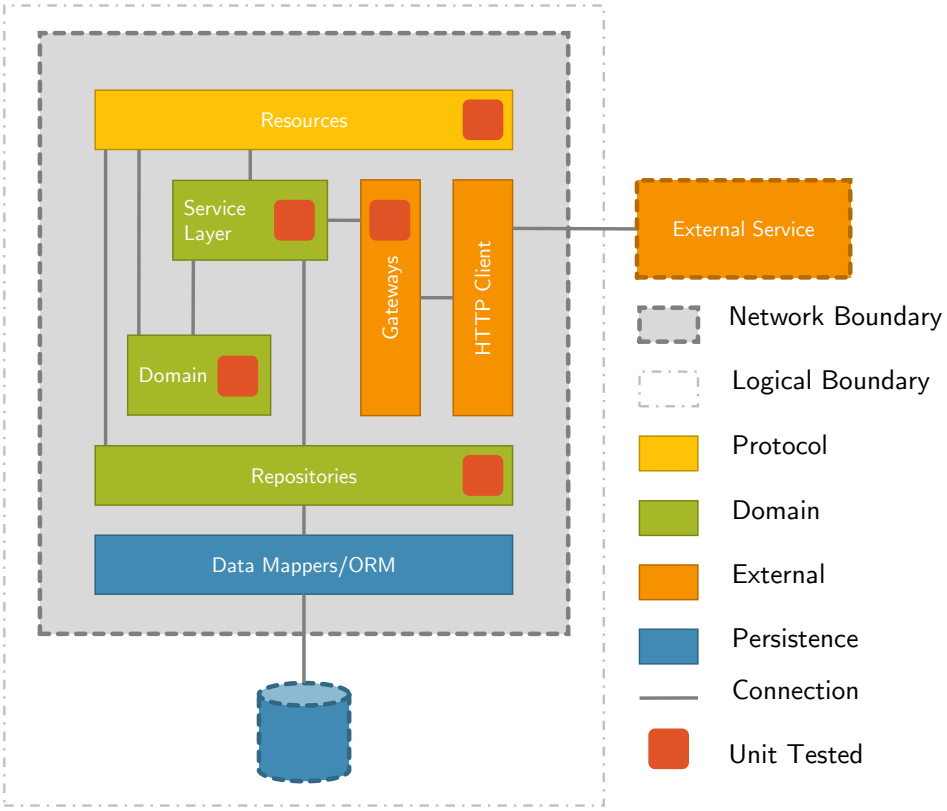

Abbildung 4: Unit Testing Scope [Cle14]

3.2.0.2 Integrationstests text

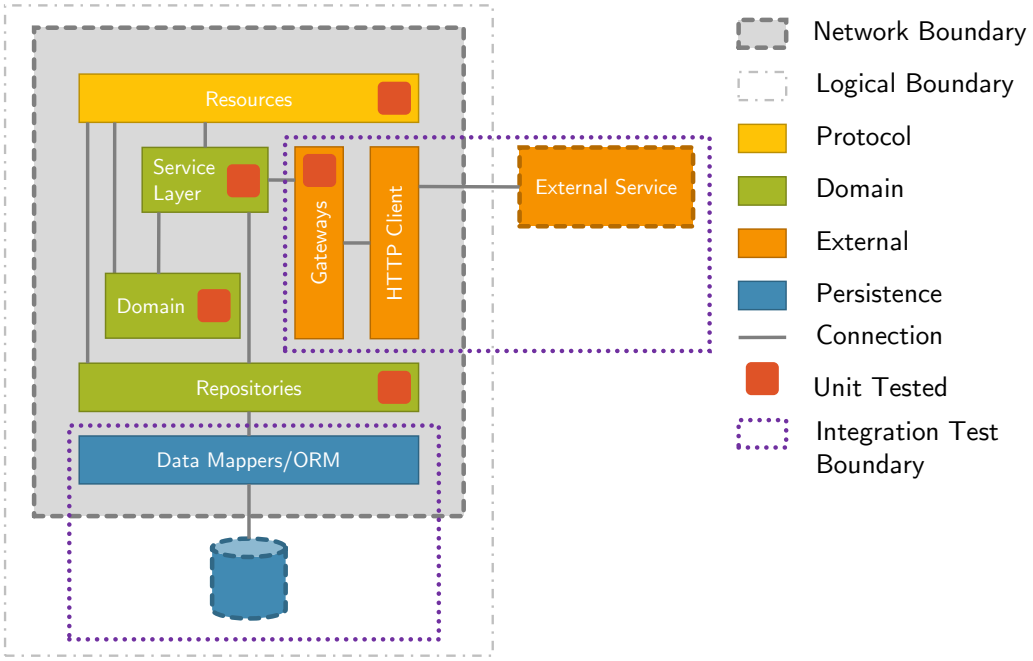

Abbildung 5: Integration Testing Scope [Cle14]

3.2.0.3 Komponententests

text

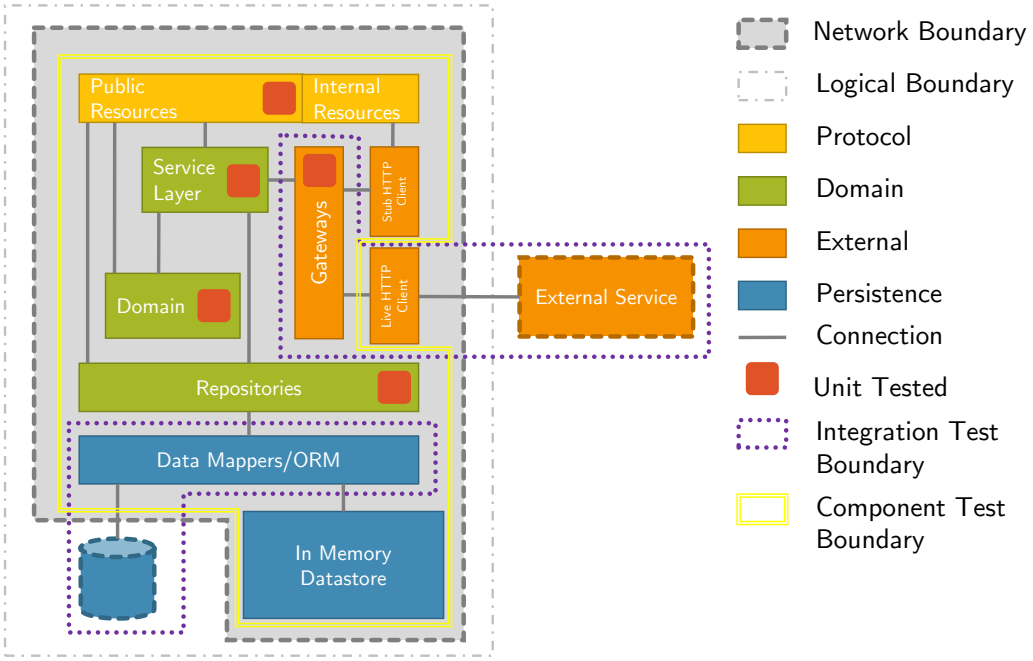

Abbildung 6: Component Testing Scope [Cle14]

3.2.0.4 Contract Testing text

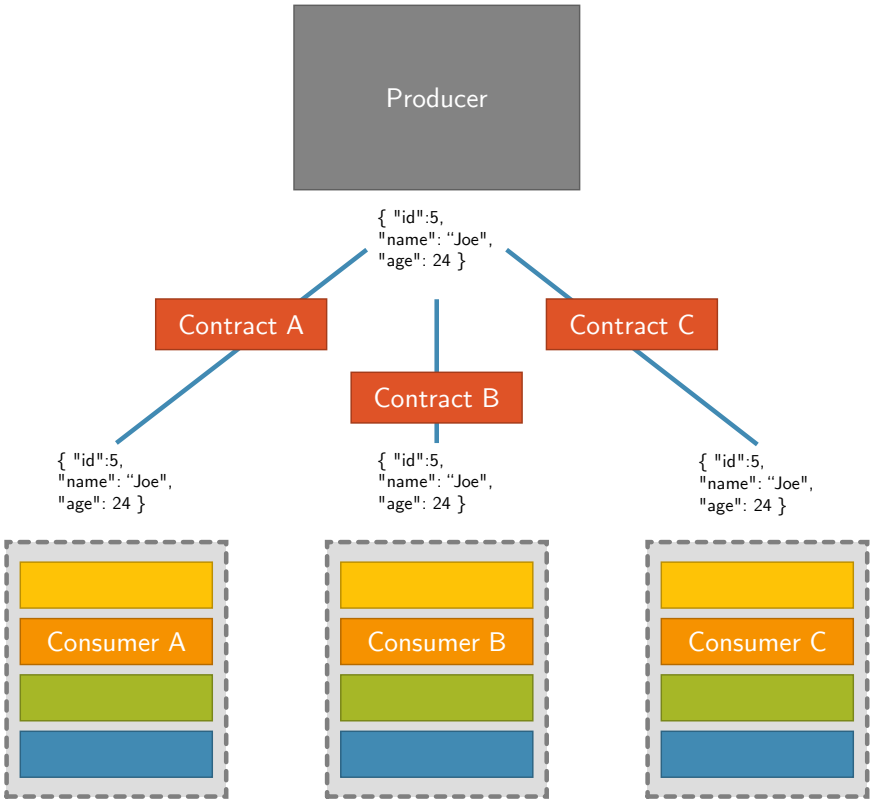

Abbildung 7: Contract Testing [Cle14]

3.2.0.5 End-To-End Testing

text

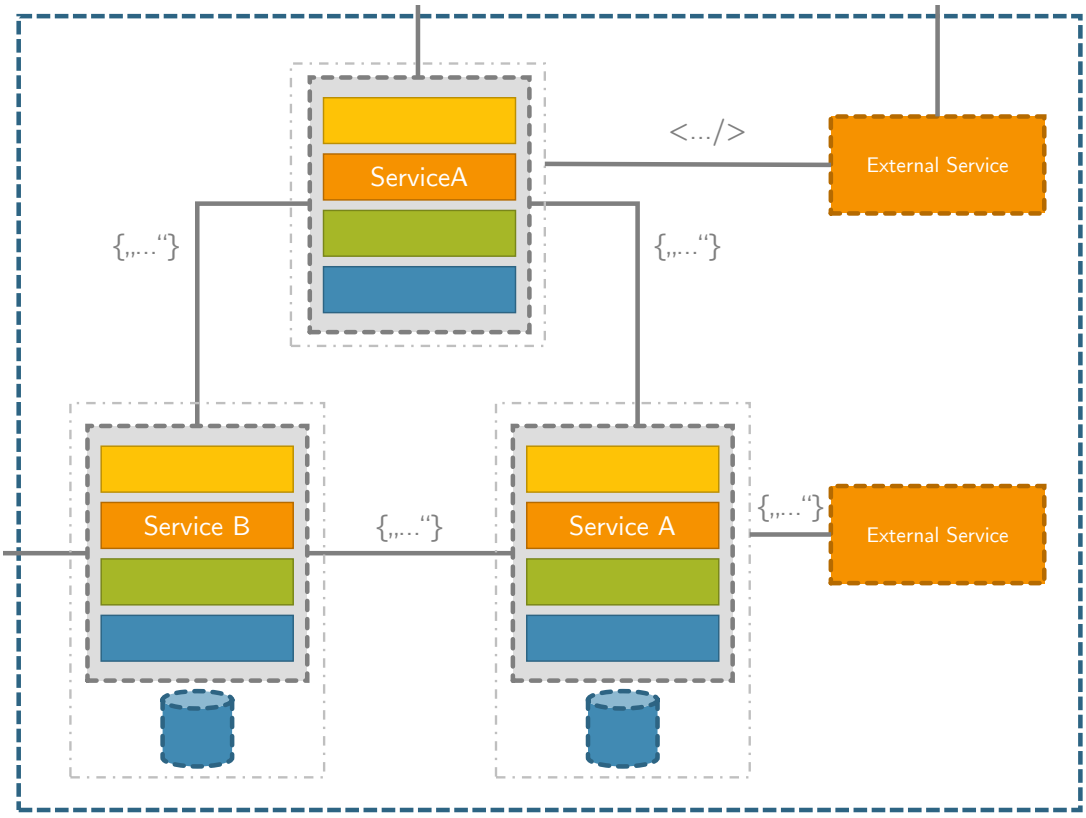

Abbildung 8: End-To-End Testing Scope [Cle14]

3.2.1 Sinnvolle Teststrategien für den generativen Ansatz

3.2.2 Frameworks zum Umsetzen von Test-Strategien

4 Anforderungen an generierte Tests

4.1 Benötigte Daten

4.2 Notwendige Änderungen/Erweiterungen von Barrakuda

5 Implementierung in Barrakuda

5.1 Referenz-System

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

[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. – 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. – Abgerufen am 12.12.2016

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

[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

[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