Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Alexander Piehl

Testkonzepte für Microservice-basierte Systeme am Beispiel des MARS Frameworks

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander Piehl

Testkonzepte für Microservice-basierte Systeme am Beispiel des MARS Frameworks

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Bettina Buth
Zweitgutachter: Prof. Dr. rer. nat. Thomas Lehmann

Eingereicht am: 11. Januar 2018

Alexander Piehl

Thema der Arbeit
Testkonzepte für Microservice-basierte Systeme am Beispiel des MARS Frameworks

Stichworte
Testen, Microservices, MARS Framework, Exploratory Testing, Consumer Driven Contract Test

Kurzzusammenfassung
Die vorliegende Masterarbeit schlägt einen Ansatz für das Testen von Microservice-basierten Systemen vor. Als Beispiel dient das an der HAW Hamburg entwickelte MARS Framework, für das die Herausforderungen herausgearbeitet werden, die im Zusammenhang mit der Architektur Microservices und Testen stehen. Dazu werden aktuelle Testkonzepte vorgestellt und miteinander verglichen. In Ergänzung zu diesen Testkonzepten wird untersucht, inwiefern die Verwendung von Exploratory Testing in diesem Kontext sinnvoll ist, insbesondere im Zusammenhang mit Systemtests. Anhand dieser Informationen und am Beispiel einer reduzierten Demo-Version vom MARS Framework wird ein Testkonzept entworfen.
Alexander Piehl

Title of the paper
Test concepts for microservice-based systems using the example of the MARS framework

Keywords
Testing, Microservices, MARS Framework, Exploratory Testing, Consumer Driven Contract Test

Abstract
This master thesis proposes an approach for testing microservice-based systems. The MARS Framework, developed at the Hamburg University of Applied Sciences (HAW Hamburg), will serve as an example for microservice-based systems. Furthermore, challenges will be discussed, which occur during the testing procedures of the microservice architecture. In addition, recent testing concepts will be evaluated. Besides presenting testing concepts, the discussion will also include a debate on whether the usage of exploratory testing is practical especially in terms of system tests. Finally, a test concept will be designed due to these evaluations and an example of a reduced demo version of the MARS Framework.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

In dieser Masterarbeit wird die Thematik des Testens von Microservice-basierten Systemen behandelt. Die Verwendung von Microservices bzw. die Verwendung dieser Architektur ist zurzeit populär. Verschiedene Unternehmen haben ihre Webservices auf Microservices umgebaut, wie z.B. Netflix, Amazon und Ebay [Fow, O'H06].

Das Merkmal der Architektur Microservices ist die Verwendung von mehreren kleineren Services, Microservices, anstatt eines monolithischen Services [Fow14]. Dabei agieren die einzelnen Microservices als eigenständige Programme und ergeben im Zusammenwirken die Gesamtanwendung. Die Architektur verfolgt dabei eine service-zentrierte Sicht. Durch diese Aufteilung sollen sich verschiedene Vorteile ergeben, wie z.B die Skalierung zu vereinfachen und die unabhängige Entwicklung der einzelnen Komponenten bzw. Microservices zu fördern.

Aufgrund des zurzeit populären Status der Architektur gibt es viele verschiedene Informationen zu einzelnen Aspekten der Architektur. Jedoch wird ein Aspekt sehr wenig oder überhaupt nicht erwähnt und zwar der Aspekt des Testens. Selbst Unternehmen wie Amazon, Netflix und etc. geben kaum Informationen darüber aus, wie sie ihre Microservice-basierte Software testen. Daraus entsteht eine Wissenslücke bei den Microservices, die gravierend ist.

Aufgrund dieser fehlenden Informationen ist es interessant zu untersuchen, was eigentlich Testen einer Microservice-basierten Anwendung bedeutet. In diesem Zuge stellen sich mehrere Fragen, wie z.B. ob die Architektur Microservices das Testen beeinflusst. Wenn ja, stellt sich auch die Frage, wie genau die Architektur das Testen beeinflusst. Daraus generiert sich wiederum die Fragestellung, wie ein Microservice-basiertes System am besten getestet wird, genauer gesagt wie das optimale Testkonzept aussieht. Schon wegen der vielen offenen Fragen ist es wert, diese Thematik näher zu untersuchen.

Die Fragen werden am Beispiel des MARS Frameworks beantwortet. Das MARS Framework ist eine Anwendung für Multi-Agenten Simulationen, die an der HAW Hamburg entwickelt wird. Als Architektur nutzt das MARS Framework Microservices [HATCea16, mar]. Das MARS Framework bietet sich besonders an, da zurzeit kein Testkonzept dafür existiert.

Neben den hier bereits ausgeführten Fragestellungen, haben sich noch weitere spezifischere Fragen ergeben. Diese werden im Kapitel Leitfragen vorgestellt. Dabei geht es unter anderem

um die Frage, wie z.B. die Herausforderungen beim Testen einer Anwendung mit der Architektur Microservices aussehen und wie ein passendes Testkonzept für MARS dazu auszusehen vermag. Das Kernmerkmal dieser Arbeit wird insbesondere die Entwicklung eines Testkonzeptes für das MARS Framework bilden. Jedoch werden auch die anderen Fragen im Laufe dieser Arbeit beantwortet werden.

Nach der Vorstellung der Leitfragen werden im Kapitel 3 die Grundlagen für diese Arbeit vorgestellt. Ohne diese Grundlagen können bestimmte Sachverhalte nicht näher beleuchtet werden. Dabei werden Grundbegriffe des Testens erläutert und die Architektur Microservices wird inklusive ihrer vermeintlichen Vorteile dargelegt. Darüber hinaus wird das MARS Framework ausführlicher vorgestellt.

Im nachfolgenden Kapitel 4 werden die Herausforderungen an das Testen einer Anwendung mit Microservices beschrieben. Es werden dabei nicht nur allgemeine Herausforderungen dargestellt, sondern auch die, die insbesondere durch das MARS Framework entstehen. Im Anschluss daran werden vier verschiedene Testkonzepte vorgestellt, von denen drei davon explizit auf das Testen von Microservices ausgelegt sind. Diese Testkonzepte werden miteinander verglichen und im Anschluss einer kurzen Bewertung hinsichtlich ihrer Einsatzbarkeit unterzogen. Zum Abschluss dieses Kapitels wird noch untersucht, ob der Einsatz von Exploratory Testing in einer Microservice-basierten Anwendung sinnvoll ist. Dabei ist Exploratory Testing sehr vereinfacht gesagt als eine besondere Form des manuellen Testens zu verstehen.

Nach dieser grundlegenden Arbeit wird im nächsten Kapitel 5 das daraus entworfene Testkonzept für das MARS Framework vorgestellt. Darauf folgend wird erläutert, warum das Testkonzept so entworfen worden ist. Dazu werden auch die Lösungsansätze für die einzelnen Herausforderungen dargelegt.

In den letzten beiden Kapiteln wird zuerst beschrieben, wie die zukünftige Arbeit an dieser Thematik aussehen kann und danach wird ein generelles Gesamtfazit über die Ergebnisse dieser Masterarbeit gezogen.

2 Leitfragen

In diesem Kapitel werden die Leitfragen aufgestellt und erläutert, die im Laufe dieser Arbeit beantwortet werden sollen. Dadurch bilden die Leitfragen den Rahmen dieser Ausarbeitung.

2.1 Was sind Microservices?

Die erste Leitfrage beschäftigt sich mit dem Thema, was Microservices sind. Damit bildet diese Leitfrage die Grundlage, auf die die weiteren Fragen aufbauen. Hierbei gilt es zu erklären, was die Charakteristika dieser Architektur sind. Dafür werden zu einem der Aufbau der Architektur vorgestellt und zum anderen die spezifischen Eigenschaften, die im Zusammenhang mit der Architektur stehen.

Ergänzend dazu muss auch untersucht werden, wie der typische Lifecycle einer Anwendung mit Microservices ausschaut. Der Lifecyle kann Einfluss auf das Testen besitzen. Insbesondere zu welchem Zeitpunkt die Tests ausgeführt werden sollen.

2.2 Was macht Microservices besonders?

Die nächste Leitfrage beschäftigt sich mit den Besonderheiten der Architektur Microservices. Da die Architektur zurzeit von großen Unternehmen wie Netflix und Amazon verwendet wird, muss es dafür Gründe geben, grade diese Architektur zu verwenden und nicht eine beliebige andere Architektur [Fow, O'H06].

Zusätzlich muss auch bei dieser Leitfrage untersucht werden, ob der dazugehörige Lifecyle Besonderheiten aufweist. Diese Besonderheiten sind wichtig zu kennen, da sie Einfluss auf das Testkonzept nehmen können.

Bei der Beantwortung dieser Leitfrage sollen die jeweiligen Besonderheiten von Microservices herausgearbeitet und vorgestellt werden. Es ist dabei wichtig zu erwähnen, dass bei der Beantwortung dieser Leitfrage keine Wertung der Besonderheiten stattfinden soll, denn eine Bewertung der Architektur Microservices ist nicht das Ziel dieser Arbeit.

2.3 Was sind die Herausforderungen an das Testen?

Ergänzend zu der Leitfrage, welche Besonderheiten die Architektur Microservices besitzt, stellt sich die nächste Leitfrage nach den Herausforderungen beim Testen. Dabei wird herausgearbeitet, welche Herausforderungen aufgrund der Architektur entstehen. Dazu wird die Beantwortung dieser Frage dahingehend ergänzt, welche besonderen Herausforderungen durch das MARS Framework entstehen. Die dabei erarbeiteten Punkte werden zusätzlich dahingehend untersucht, ob sie ausschließlich bei der Architektur Microservices auftauchen oder bereits bekannte Herausforderungen sind.

2.4 Wie gehe ich mit Änderungen um?

Bei der Erarbeitung des Testkonzeptes soll die Leitfrage beantwortet werden, wie mit Änderungen am Produkt umgegangen werden soll. Mit der Leitfrage sind alle Änderungen gemeint, von den Änderungen am Sourcecode bis hin zur Änderungen in der Struktur der Anwendung. Unter der generellen Frage, wie man mit Änderungen umgehen soll, stellen sich weitere Fragen. Dazu gehört die Frage, ob bei Änderungen bestimmte Sachverhalte berücksichtigt werden müssen, damit die Anwendungen und besonders die Änderungen testbar bleiben bzw. sind. Daher stellt sich auch die Frage, wie die Tests entworfen werden müssen, damit sie trotz Änderungen weiterhin funktionieren.

Grundsätzlich gehört auch die Frage dazu, ob unterschiedliche Arten von Änderungen existieren. Sollte es diese geben, folgt daraus die Fragen, nach Klassifizierung der Änderungen. Schlussendlich stellt sich die Frage, welche Rolle die Entkopplung spielt und welche Auswirkungen sie auf Änderungen und das Testen hat. Warum diese Frage interessant ist, wird im nächsten Kapitel deutlicher, indem die Architektur Microservices ausführlicher vorgestellt wird.

2.5 Wie sieht ein Testkonzept für MARS aus?

Unter dieser Fragestellung soll erläutert werden, wie ein optimales Testkonzept für das MARS Framework auszusehen hat.

Das Testkonzept soll sich dabei ausschließlich auf die Komponenten beschränken, die nicht für die Simulation genutzt werden. Für die anderen Komponenten soll das Testkonzept nicht gelten.

Zusätzlich zu dieser Einschränkung, wird eine weitere Einschränkung vorgenommen. Das in dieser Arbeit entwickelte Testkonzept wird sich nur auf funktionale Anforderungen konzentrieren.

Das dabei entstehende Testkonzept soll nicht nur die vorherige Leitfrage zum Thema Änderungen beantworten, sondern weitere Fragen. Dazu gehört die Frage, welche Arten von Tests existieren sollen. Darauf aufbauend entsteht die Frage, wie viele Tests einer Art es in Relation zu den anderen Testarten geben soll.

Des Weiteren soll das Testkonzept Antworten auf spezifische Herausforderungen der Architektur Microservices und spezifische Herausforderungen des MARS Frameworks geben. Dazu sollen bereits existierende Testkonzepte untersucht werden.

3 Grundlagen

In diesem Kapitel werden die Grundlagen für die vorliegende Arbeit vorgestellt. Dafür werden die Grundbegriffe des Testens vorgestellt und wie sie im Kontext dieser Arbeit definiert werden, weil von manchen Begriffen verschiedene Definitionen existieren, die in Abhängigkeit vom Kontext variieren können.

In diesem Kapitel werden auch die im praktischen Teil verwendeten Technologien wie Selenium und PACT vorgestellt. Dazu wird der von PACT implementierte Testansatz Consumer Driven Contract Test erläutert. Auch wird es eine Einführung in die Methode Exploratory Testing geben, deren Nutzen im praktischen Teil dieser Arbeit untersucht wird.

Zusätzlich wird die Architektur Microservices mit ihren Charakteristika vorgestellt. Dazu ergänzend werden auch die Besonderheiten und Vorteile der Architektur beschrieben. Da für die Erstellung eines Testkonzeptes auch der Prozess des Erstellens und Veröffentlichen der Anwendung von Bedeutung sind, wird in diesem Kapitel ein typischer Prozessablauf für Microservices beschrieben. Zum Abschluss dieses Kapitels wird das MARS Framework vorgestellt.

3.1 Grundlagen zum Testen

In diesem Abschnitt sollen verschiedene Begriffe und Technologien zum Bereich des Testens vorgestellt werden, die für die weitere Arbeit von Bedeutung sind.

Dafür wird zunächst ein kurzer Überblick über die Grundbegriffe gegeben und wie sie definiert werden. Darauf folgend werden die Technologien Selenium und PACT vorgestellt, besonders Selenium ist für den praktischen Teil dieser Arbeit von Bedeutung. Dazu passend wird auch das Exploratory Testing vorgestellt.

3.1.1 Grundbegriffe

Die Definitionen für die Begriffe, die in dieser Arbeit zum Thema Testen verwendet werden, sind zum größten Teil die Definitionen aus dem ISTQB-Standard [AS12]. Bei der Vorstellung der

Testkonzepte kann es zu Abweichungen kommen, da dort die Definitionen von den jeweiligen Autoren der Testkonzepte übernommen werden.

Testarten

Wie im ISTQB-Standard wird auch hier nach verschiedenen Testarten unterschieden. Es gibt funktionale Tests, nicht-funktionale Tests und Regressionstests.

Bei funktionalen Tests wird getestet, ob die funktionalen Anforderungen erfüllt sind. Funktionale Anforderungen beschreiben, wie sich das System verhalten soll. Eine funktionale Anforderung könnte sein, dass der Nutzer die Möglichkeit besitzt, sich ein Kundenkonto anzulegen. Die Erfüllung dieser funktionalen Anforderungen bildet die Grundlage für einen produktiven Einsatz der Anwendung [AS12].

Im Gegensatz dazu stehen die nicht-funktionalen Tests. Hierbei werden die nicht-funktionalen Anforderungen getestet. Nicht-funktionale Anforderungen beschreiben die Qualität der Anwendung. Dazu gehören verschiedene Punkte, wie z.B. Performanz, Sicherheit und Usability. Ein Beispiel dafür ist, dass eine Antwort von einem Server nicht länger als zehn Sekunden dauern darf [AS12].

Regressionstest beinhalten sowohl funktionale als auch nicht-funktionale Tests. Es wird getestet, ob wiederholte Tests dieselben Ergebnisse liefern. Damit wird sichergestellt, dass Änderungen nicht zu Fehlern führen [AS12].

Teststufen

Es existieren verschiedene Teststufen. Eine Teststufe ist der Komponententest. Der Komponententest wird auch als Unit-Test bezeichnet. Dieser Begriff wird auch im weiteren Verlauf dieser Arbeit verwendet.

Beim Unit-Test wird die kleinste Softwareeinheit getestet. Die kleinste Softwareeinheit ist abhängig von der gewählten Programmiersprache. In der Programmiersprache Java wird mit einem Unit-Test z.B. eine Methode getestet. Hierbei handelt es sich um einen funktionalen Test, denn es wird getestet, ob die Komponente ihre funktionalen Anforderungen erfüllt. Erfüllt sie ihre Anforderungen, gilt der Test als erfolgreich [AS12].

Die nächste Teststufe ist der Integrationstest. Beim Integrationstest wird das Zusammenwirken der verschiedenen Komponenten getestet. Integrationstests haben das Ziel, Fehler in den Schnittstellen der einzelnen Komponenten zu finden, die eine Zusammenarbeit der Komponenten verhindern [AS12].

Nach den Integrationstests folgt der Systemtest. Bei den Systemtests wird geprüft, ob die Anforderungen an das komplette System bzw. Programm erfüllt werden. Dabei werden funk-

tionale sowie nicht-funktionale Anforderungen getestet [AS12]. Auf dieser Stufe wird z.B. getestet, ob der Nutzer ein Kundenkonto anlegen kann.

3.1.2 Selenium

Mit dem Framework Selenium können Browser automatisiert gesteuert werden [Gun15, Bur12]. Diese Funktion wird hauptsächlich zum Testen von webbasierten Anwendungen verwendet. Es können Interaktionen definiert werden, die im Browser ausgeführt werden sollen. Aufgrund der Automatisierung können die Interaktionen und somit die Tests wiederholt ablaufen. Daher bietet sich das Framework Selenium sehr gut für Regressionstests von Weboberflächen an. [sel]

Mit Selenium können die gängigsten Browser wie Firefox, Chrome und Internet Explorer gesteuert werden [Gun15]. Die Steuerung findet über Skripte statt, die mit verschiedenen Programmiersprachen entwickelt werden können [sel, Bur12]

Das Framework Selenium wird in zwei Produkte unterschieden. Es gibt die Selenium IDE und den Selenium WebDriver [sel, Bur12]. Selenium IDE ist ein Add-on für den Browser Firefox. Damit können Interaktionen mit dem Browser aufgezeichnet werden. Aus diesen Aufzeichnungen können automatisiert Skripte erstellt werden und wiederholt ausgeführt werden [sel, Bur12]. Zusätzlich können über die Oberfläche des Add-ons weitere Aktionen hinzugefügt werden. Allerdings kann mit der Selenium IDE nur der Firefox Browser gesteuert werden. In Abbildung 3.1 ist ein Beispiel Testfall mit Selenium IDE abgebildet. Bei diesem Testfall wird die Seite des Departments Informatik an der HAW Hamburg geladen und geprüft, ob die Überschrift vorhanden ist.

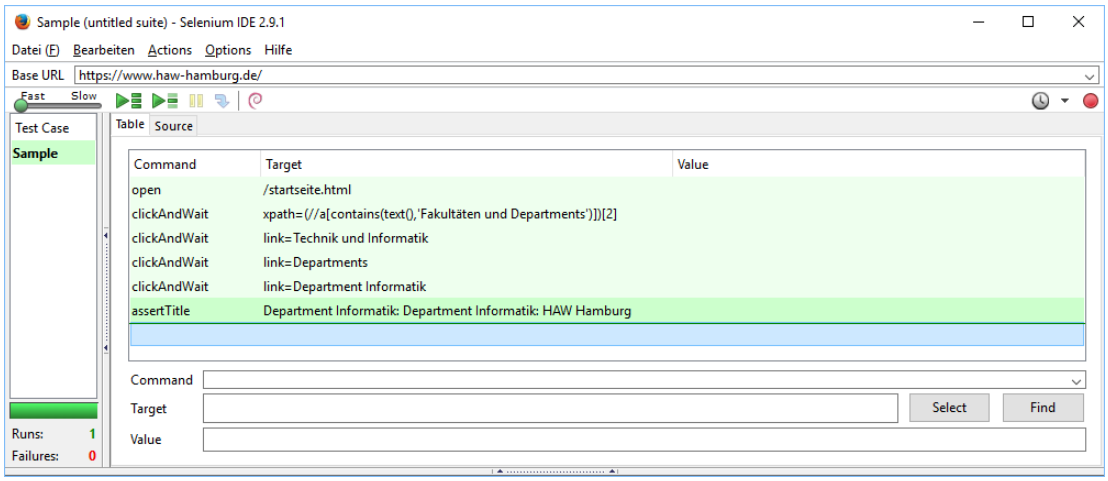

Abbildung 3.1: Beispiel Testfall mit Selenium IDE

Beim Selenium WebDriver existiert keine Oberfläche. Dafür ist der Selenium WebDriver nicht auf den Firefox Browser beschränkt. Wie in der Einleitung beschrieben, können die gängigsten Browser gesteuert werden. Dafür werden verschiedene WebDriver benötigt, um die unterschiedlichen Browser anzusprechen. Somit existieren unter anderem WebDriver für Chrome, Firefox und Internet Explorer. Die WebDriver dienen ausschließlich der Ansteuerung der Browser. Auf die Implementierung der Skripte haben sie keinen Einfluss [sel].

Die Abbildung 3.2 zeigt denselben Testfall als Skript in Java an, wie das Beispiel mit Selenium IDE.

```
@Test
public void sampleTest(){
    WebDriver driver = new FirefoxDriver();
    driver.manage().window().maximize();
    driver.get("https://www.haw-hamburg.de");
    driver.manage().timeouts().implicitlyWait( time: 10, TimeUnit.SECONDS);

    driver.findElement(By.xpath("(//a[contains(text(),'Fakultäten und Departments')])[2]")).click();
    driver.findElement(By.linkText("Technik und Informatik")).click();
    driver.findElement(By.linkText("Departments")).click();
    driver.findElement(By.linkText("Department Informatik")).click();

    String actualTitle = driver.getTitle();
    String expectedTitle = "Department Informatik: Department Informatik: HAW Hamburg";
    Assert.assertEquals( message: "Fehler Titel stimmt nicht.",actualTitle,expectedTitle);
}
```

Abbildung 3.2: Beispiel Testfall mit Selenium WebDriver

Innerhalb dieser Arbeit wird ausschließlich der Selenium WebDriver verwendet. Mit dem Selenium WebDriver können komplexere Skripte entworfen werden, die für die Implementierung der Testfälle im praktischen Abschnitt benötigt werden. Darüber hinaus ist die Selenium IDE mehr für kurze Bug Tests und die Aufzeichnung von manuellen Tests geeignet [sel].

Eine ausführliche Beschreibung des Frameworks und eine Einführung in die Grundlagen finden sich in der Ausarbeitung zum Grundprojekt [Pie17b].

3.1.3 PACT

PACT ist eine Implementierung des Testansatzes Consumer Driven Contract Test, welcher im nächsten Abschnitt erläutert wird.

Das Framework PACT ist open-source und wird auf GitHub verwaltet. Mithilfe dieses Tools können Schnittstellen bzw. Kontrakte von Services getestet werden. Die Tests können in verschiedenen Sprachen wie Java und C# geschrieben werden [pac17].

Eine Beschreibung des Ablaufes und der genauen Tests bei PACT sind in der Ausarbeitung zum Hauptprojekt dargelegt wurden [Pie17a].

3.1.4 Consumer Driven Contract Test

Consumer Driven Contract Test, kurz CDCT, wurden bereits im Hauptprojekt ausführlich behandelt [Pie17a]. Daher wird die Begrifflichkeit Consumer Driven Contract Test nur kurz erläutert.

Das Ziel von CDCT ist zu testen, ob die Verträge für die Kommunikation zwischen den Services eingehalten werden. Im Prinzip beschreibt jede Schnittstelle auch einen Vertrag.

Dabei enthalten die Verträge Informationen über die zur Verfügung stehenden Operationen und welche Parameter dafür benötigt werden. Ergänzend dazu enthalten die Verträge auch Informationen darüber, wie die Antworten auszusehen haben und in welchen Dateienformaten sie zur Verfügung stehen.

Consumer Driven Contract ist dabei eine spezielle Form der Verträge. Consumer Driven Contract ist eine Sammlung von mehreren Consumer Contracts. Ein Consumer Contract enthält die Funktionen, die ein Service (Consumer) an einem anderen Service (Provider) nutzen möchte. Da es mehrere Consumer pro Provider geben kann, können auch mehrere Consumer Contracts existieren. Dies ist in Abbildung 3.3 dargestellt.

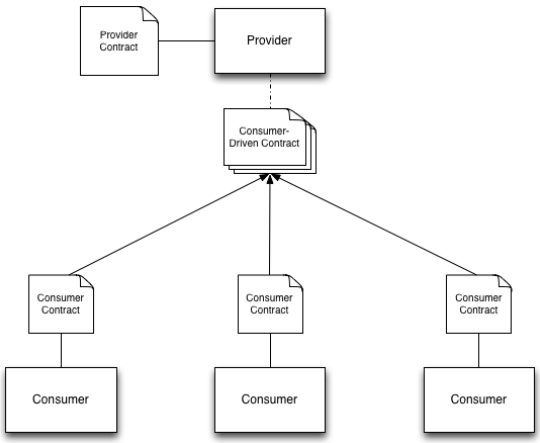

Abbildung 3.3: Consumer Driven Contracts [BS15]

Wie schon beschrieben ist die Aufgabe von CDCT die Einhaltung der gesammelten Consumer Contracts zu testen. Dafür wird im ersten Schritt getestet, ob der Consumer den Vertrag einhält. Dieser stellt seine Anfrage an einen simulierten Provider, welcher mit dem im Vertrag definierten Antworten eine Response bildet. Werden die Anfragen so gestellt, wie im Vertrag stehend, und kann der Consumer die Antworten verarbeiten, gilt dieser Schritt als erfolgreich. Sollte dies erfolgreich sein, werden simulierte Anfragen an den Provider gestellt. Antwortet der

Provider wie im Vertrag definiert, gilt der Test als erfolgreich. Das Vorgehen ist in Abbildung 3.4 abgebildet.

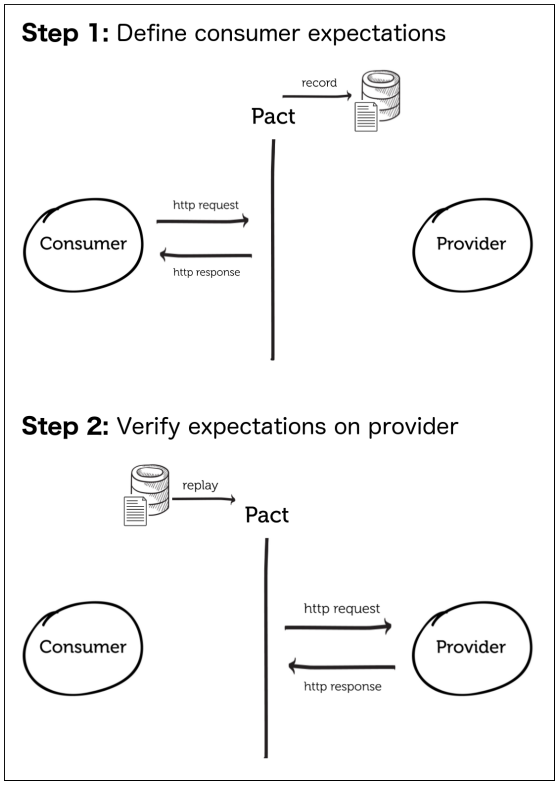

Abbildung 3.4: CDCT mit PACT [pac17]

Der Vorteil von CDCT ist es, Regressionstests auf die Schnittstellen ausführen zu können. Zudem soll CDCT in der Lage sein, Integrationstests zu ersetzen, da CDCT die Einhaltung der Verträge und damit auch das Zusammenarbeiten der Services testet.

Zusätzlich kann CDCT auch bei gewollten Änderungen hilfreich sein. Muss z.B. eine Änderung an einem Provider vorgenommen werden, kann durch das Ausführen von CDCT festgestellt werden, welche Consumer angepasst werden müssen. Bei diesen würden die Tests dann fehlschlagen.

3.1.5 Exploratory Testing

Exploratory Testing fällt in den Bereich des manuellen Testens. Dabei ist es definiert als das gleichzeitige Kennenlernen der Software sowie die Erstellung und Ausführung von Tests [SBF14].

Dieser Ansatz unterscheidet sich vom gängigen Testansatz, bei dem das Entwerfen der Tests und das Ausführen zwei verschiedene Schritte sind. Der Tester lernt das Programm durch das Ausführen der Tests immer besser kennen. Dabei werden Tests erstellt und gleichzeitig ausgeführt. Hierbei werden die Tests nicht aus den Anforderungen hergeleitet. Da es sich um manuelles Testen handelt, gibt es keine geskripteten Tests. Wie aus der vorherigen Erklärung abgeleitet werden kann, werden die Tests dynamisch entworfen, ausgeführt und auch modifiziert. Eine Beschränkung auf eine bestimmte Teststufe existiert nicht [SBF14, IR05].

Vereinfacht gesagt, interagieren die Tester mit der Anwendung, wie sie möchten und nutzen sowohl die Informationen, die sie bereits besitzen, als auch die aus der Anwendung ablesbaren. Dabei werden die Funktionen der Anwendung ohne Einschränkung getestet [Whi09]

Um diese Entscheidungen treffen zu können, benötigt der Tester grundlegendes Wissen über die Anwendung bzw. Erfahrung mit der Anwendung. Der Erfolg des Testens ist abhängig vom Wissen des Testers [SBF14]. Dahingehend kann dieser Ansatz bei erfahrenen und gut qualifizierten Testern sehr erfolgversprechend sein [Whi09].

Das dafür benötigte Wissen kann aus mehreren verschiedenen Quellen besorgt werden. Zum einem aus der Dokumentation der Anwendung und aus der Beobachtung der Anwendung während des Testens [SBF14]. Ergänzend dazu ist es hilfreich, sich bereits vor dem Testen sehr gut mit der Anwendungsdomäne, mit der Plattform und/oder mit den häufigsten Fehlerursachen in der Anwendungsdomäne auszukennen [SBF14].

Whittaker betont die Notwendigkeit eines erweiterten Vorwissens über die Anwendung, die zu testen ist. Er vergleicht dies mit einem Stadtrundgang durch eine unbekannte Stadt. Zwar könnten auch ohne Plan Sehenswürdigkeiten entdeckt werden, jedoch geschehe dies eher zufällig und einige Sehenswürdigkeiten würden verpasst werden. Daher ist für ihn unumgänglich einen Plan zu haben, wie vorgegangen werden soll [Whi09].

Exploratory Testing wird als Ergänzung zu den bisherigen Tests gesehen und nicht als deren Ersatz. Mit diesem Testansatz sollen besonders Bereiche getestet werden, die bisher unbekannt sind [IR05].

Die Verwendung von Exploratory Testing schließt die Verwendung von Tools nicht aus, die das automatisierte Testen unterstützen. Dabei können Programme verwendet werden, welche den Bildschirm oder die Tastatureingaben aufzeichnen [Whi09, WAC12]. Aus den Aufzeichnungen können anschließend automatisierte Tests erstellt werden.

Nach Whittaker können Tests in Exploratory Testing in zwei Unterbereiche eingeteilt werden. Ein Bereich sind die Small Tests und der andere Bereich die Large Tests. Bei Small Tests werden innerhalb eines einzelnen Tests lokale Entscheidungen getroffen, z.B. welche Daten eingegeben

werden. Im Gegenzug wird der Tester bei Large Tests beim Entwerfen von Testplänen und Teststrategien unterstützt [Whi09].

Generell bietet sich der Ansatz Exploratory Testing sehr gut für agile Prozesse an, da die Prozesse meist sehr kurz sind und die Notwendigkeit der Anpassung der Tests bei Exploratory Testing wegfällt [Whi09].

3.2 Microservices

In einer vorherigen Ausarbeitung zum Grundprojekt [Pie17b] wurde die Architektur Microservices nur kurz skizziert. Daher wird hier die Architektur ausführlicher erläutert.

Wie der Einleitung entnommen werden kann, ist das Gegenstück zu einem Microservice-basierten Systems ein monolithisches System, mit fest verankerten Services. Deswegen bietet es sich an, Microservices im Vergleich mit einem monolithischen Services zu erklären. Die Grundlage dafür bildet der Vergleich zwischen den beiden Systemen von Martin Fowler [Fow14].

Ein monolithischer Webservice erfüllt alle Anforderungen innerhalb eines geschlossenes Programmsystems. Bei der Architektur Microservices werden die einzelnen Aufgaben in eigenständige Prozesse bzw. Programme ausgegliedert [Fow14, SR15, SRT15, Wol15].

Es findet damit eine Modularisierung der Anwendung statt. Dabei soll jeder Microservice nur genau eine Aufgabe erfüllen. In dem Sinne wird das aus der objektorientierten Programmierung bekannte Paradigma der Single-Responsibility auf die Ebene der Services übertragen. Die Funktionen des Gesamtsystems ergeben sich aus dem Zusammenwirken der einzelnen Services.

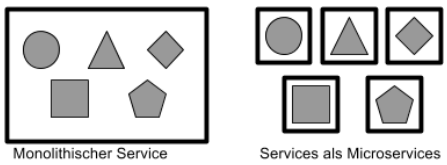

Abbildung 3.5: Vergleich monolithischer Service und Microservices [Fow14]

Zwischen den einzelnen Modulen bzw. Services existiert nur eine lose Kopplung. Damit ist gemeint, dass die jeweiligen Abhängigkeiten zwischen den einzelnen Services gering sind. Dadurch wirken sich Änderungen nur lokal aus und betreffen nicht das gesamte System [Wol15, Cle14].

Auch auf die Kommunikation hat die Unabhängigkeit Auswirkungen. Die Kommunikation findet auf der Ebene des Netzwerkes über das HTTP-Protokoll statt, sowohl lokal als auch

global. Wegen der Entkopplung der einzelnen Services werden meist Kommunikationsarten verwendet, welche dies unterstützen. Ein Beispiel dafür ist REST [Fow14, Wol15].

Jeder Service muss eine definierte Schnittstelle besitzen. Nur hierdurch ist es gewährleistet, dass die Services kommunizieren können. Ohne Kommunikation mit den anderen Microservices ist eine Zusammenarbeit nicht möglich. Da die Microservices nur Teilaufgaben übernehmen, ist eine Zusammenarbeit unumgänglich. Ansonsten können die Aufgaben des Gesamtsystems ggf. nicht bewältigt werden [New15].

Aufgrund der strikten Aufgabentrennung und der losen Kopplung sind die Services unabhängig voneinander bis auf die Schnittstellen zueinander. Dadurch ergeben sich verschiedene Freiheiten für die Services, wie z.B. der Prozess zum Erstellen und Veröffentlichen der Services. Darüber hinaus bietet sich die Möglichkeit, die verwendete Technologie für den Services frei zu wählen. Verschiedene Services können mit unterschiedlichen Programmiersprachen entwickelt werden und unterschiedliche Datenbanksysteme verwenden [Wol15, New15].

Da bei der Architektur Microservices die Aufgabenverteilung nicht mehr über Objekte geschieht, sondern über unabhängige Services, verlagert sich die Komplexität der Gesamtanwendung auf eine andere Ebene. Wie schon beschrieben, kommunizieren die Services über das Netzwerk. Damit verlagert sich die Komplexität der Anwendung vom internen Code-Bereich hinaus in das Netzwerk [Wol15].

Damit sind die Grundcharakteristika von Microservices erläutert. Im nächsten Abschnitt werden die Vorteile von Microservices vorgestellt.

3.2.1 Vorteile

In diesem Abschnitt werden die Vorteile der Architektur Microservices vorgestellt. In einer vorausgehenden Arbeit wurden die Vorteile der Architektur bereits vorgestellt [Pie16]. Dieser Abschnitt basiert zu einem auf die vorausgehende Arbeit und ergänzt zusätzlich neue Aspekte.

Es ist dabei wichtig anzumerken, dass die Vorteile sich auf ein ideales System beziehen, also eine Anwendung, in der die Architektur Microservices perfekt umgesetzt worden ist.

Die meisten Vorteile von Microservices lassen sich aus den Grundcharakteristika ablesen. Der größte Vorteil von Microservices ist die sehr starke Modularisierung der einzelnen Komponenten [Wol15]. Dadurch können Services und besonders deren Funktionen für verschiedene Szenarien wiederverwendet werden. Dazu bietet die Modularisierung die Möglichkeit, dynamischer und zeitsparender auf Veränderungen zu reagieren [New15].

Die starke Modularisierung und die damit verbundene Unabhängigkeit sind Grundlage für verschiedene Vorteile der Architektur Microservices.

Die Unabhängigkeit der Services ermöglicht ein ebenfalls unabhängiges Erstellen und Veröffentlichen. Während bei einem monolithischen Service immer der gesamte Service deployt werden muss, können bei der Architektur Microservices die Services einzeln veröffentlicht werden [Wol15, New15].

Daraus ergeben sich zwei Vorteile. Bei Änderungen an einem Service muss nur dieser neu veröffentlicht werden und die anderen Services können unberührt bleiben. Bei einem monolithischen System muss dagegen bei Änderungen immer das komplette System neu erstellt und veröffentlicht werden [New15].

Der zweite Vorteil ist die Möglichkeit, Services erst zu veröffentlichen, wenn sie gebraucht werden. Damit kann unnötiger Ressourcenverbrauch reduziert werden.

Microservices sind leichter zu ersetzen als ein monolithisches System. Einmal im Bezug auf neue Versionen und zum anderen durch einen neu implementierten Microservice, der den alten Microservice ersetzt [Wol15, New16, New15].

Die leichte Ersetzbarkeit von Services ermöglicht eine nachhaltige Softwareentwicklung. Durch die strikte Modularisierung und der Kommunikation über die Schnittstellen können nur bewusst gewollte Abhängigkeiten zu anderen Services entstehen [WSH16]. Services, die eine veraltete Technologie verwenden oder nur schwer zu warten sind, können leicht durch einen neuen Service ersetzt werden [Wol15, New15].

Wie im obigen Abschnitt beschrieben, kann für jeden Microservice eigenständig entschieden werden, welche Technologie eingesetzt werden soll. Nur bei der Technologie für die Kommunikation muss Einigkeit herrschen. Dadurch ergibt sich der Vorteil immer die Technologie zu verwenden, die am geeignetsten für die Aufgabe des Services ist [Fow14, Wol15, New16, New15].

Microservices bieten zudem den Vorteil, die betriebliche Abstimmung zu verringern. Es müssen nur die Technologien für die Schnittstellen untereinander abgestimmt werden. Die weiteren Aspekte, wie die Wahl der Programmiersprache, sowie das Erstellen und Produktivsetzen von Services, können, wie bereits beschrieben, unabhängig voneinander geschehen [New15, WSH16].

Die Architektur bietet eine höhere Belastbarkeit. Der Ausfall eines Service beschränkt sich nur auf diesen einen Service. Andere Services erkennen diesen Ausfall und isolieren den ausgefallenen Service. Dies steht im Gegensatz zu einem monolithischen System, bei dem der Ausfall eines Moduls zu einem kompletten Ausfall des Systems führt [New15].

Bei der Skalierung der Anwendung Microservices existiert ein weiterer Vorteil der Architektur Microservices. Aufgrund der Modularisierung und der damit verbundenen Unabhängigkeit müssen nur die Services skaliert werden, bei denen es notwendig ist. Bei einer monolithischen Anwendung sieht es dagegen anders aus. Hier muss immer das komplette

System skaliert werden. Dies kann zur Folge haben, dass auch Komponenten skaliert werden, bei denen es eigentlich nicht notwendig ist. Dadurch kann zusätzliche Arbeit entstehen [New16, Wol15, Fow16, New15].

Hiermit sind die Vorteile der Architektur Microservices dargelegt. Im nächsten Abschnitt wird ein exemplarischer Build-Prozess für einen Microservice vorgestellt.

3.2.2 Build Prozess Microservices

Microservices werden meist in Form von “Continuous Delivery" erstellt und für die Produktion bereitgestellt [Wol15, New16]. “Continuous Delivery" ist ein Prozess, der die kontinuierliche Bereitstellung von neuen Versionen für die Produktion abbildet und wird meist in Kombination mit anderen agilen Methoden eingesetzt.

Zentrale Technologie von “Continuous Delivery" ist die ”Deployment Pipeline" [JH10, HSS16]. Die ”Deployment Pipeline" begleitet den Prozess von der Änderung bis hin zum Veröffentlichen der neuen Version. Dabei enthält die Pipeline verschiedene Schritte. Diese können das Erstellen der Anwendung, das Veröffentlichen und das Testen sein. Die genauen Stationen und deren Definitionen hängen von der Anwendung ab, welche die Pipeline durchlaufen [JH10]. Der Übergang zur nächsten Station erfolgt nur, wenn die vorherige Station erfolgreich abgeschlossen wurde. Die Kriterien, ob eine Station erfolgreich ist, sind abhängig von der Station und der Anwendung [JH10].

Die Pipeline läuft vollständig automatisiert ab und bedarf keines manuellen Eingriffes. Mit jeder Änderung wird die Pipeline gestartet und der Prozess ausgeführt [JH10]. Daher auch der Begriff “Continuous" , welcher mit fortlaufend und kontinuierlich übersetzt werden kann.

Da jeder Microservice ein unabhängiges Programm ist, haben die Microservices häufig jeder eine eigene “Continuous Delivery" Pipeline [Wol15, New16, HSS16]. Für einen beispielhaften Microservice könnte eine Pipeline wie in Abbildung 3.6 aussehen [FL15, Wol15, New16].

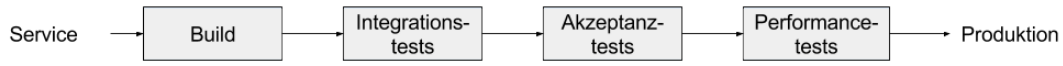

Abbildung 3.6: Beispiel Pipeline für einen Microservice [FL15, Wol15, New16]

Die in der Abbildung 3.6 dargestellten Stationen erfüllen die verschiedenen Aufgaben. Bei der Station “Build" werden neben dem Kompilieren des Sourcecodes auch die Unit- und Integrationstests ausgeführt. Im Anschluss daran wird der neu kompilierte Service veröffentlicht und somit für die weiteren Stationen bereit gestellt. Danach kann die Station der Integrationstests

ausgeführt werden, bei der die Zusammenarbeit der Services geprüft wird. Darauf folgend wird die Station der Akzeptanztests ausgeführt. Danach werden Performancetests ausgeführt. Als letzte Station folgt die Produktivsetzung der neuen Version des Services.

Als Grundlage für die Pipeline werden verschiedene Werkzeuge benötigt. Es wird zu einem ein Software-Configuration-Management System (SCM) benötigt, wie z.B. GIT es anbietet und eine Anwendung, in der die Deployment Pipelines erstellt, gewartet und durchgeführt werden [HSS16]. Ein Beispiel dafür ist die Anwendung Jenkins. Zusätzlich sollte es eine Verbindung zwischen beiden Werkzeugen geben, damit bei Änderungen die Pipeline automatisch angestoßen wird.

3.3 MARS Framework

Das MARS Framework wird innerhalb einer gleichnamigen Forschungsgruppe an der HAW Hamburg entwickelt. Das Forschungsprojekt wird geleitet von Professor Clemen [mar, HATCea16].

Der Begriff MARS ist eine Abkürzung für Multi-Agent Research and Simulation. Mit dem Framework können Multi-Agenten Simulationen erstellt und ausgeführt werden. Generell existieren keine Einschränkungen, welche Simulationen durchgeführt werden können. Zur Zeit ist die Simulation eines Ökosystems einer Savanne am ausgereiftesten [mar].

Grundlage für die Simulation bilden Modelle, die der Nutzer selber bereitstellen muss. Ergänzend dazu müssen auch die notwendigen Daten für die Modelle zur Verfügung gestellt werden. Über die Modelle können die zu simulierenden Szenarien definiert werden. Die Ergebnisse der durchgeführten Simulationen können visualisiert werden[mar].

Das Framework MARS kann über eine Webseite bedient werden. Es versteht sich dabei als “Software as a Service" . Damit ist gemeint, dass die Anwendung vom Anbieter für die Kunden bereitgestellt wird. Der Kunde installiert die Anwendung nicht auf seinen eigenen Servern bzw. Computern [HATCea16].

In der Abbildung 3.7 ist die Startseite der MARS Webseite abgebildet. Diese Startseite ist der Ausgangspunkt für das Erstellen und Ausführen von Simulationen. Die Seite für das Hochladen der notwendigen Daten und Dateien ist in der Abbildung 3.8 dargestellt. Auf dieser Seite wird der Import gestartet.

3.3.1 Import

Viele der Untersuchungen im MARS Framework werden anhand des Import-Bereiches vorgenommen. Dazu betreffen besonders die Untersuchungen aus dem Hauptprojekt zum Thema Consumer Driven Contract Tests diesen Bereich [Pie17a].

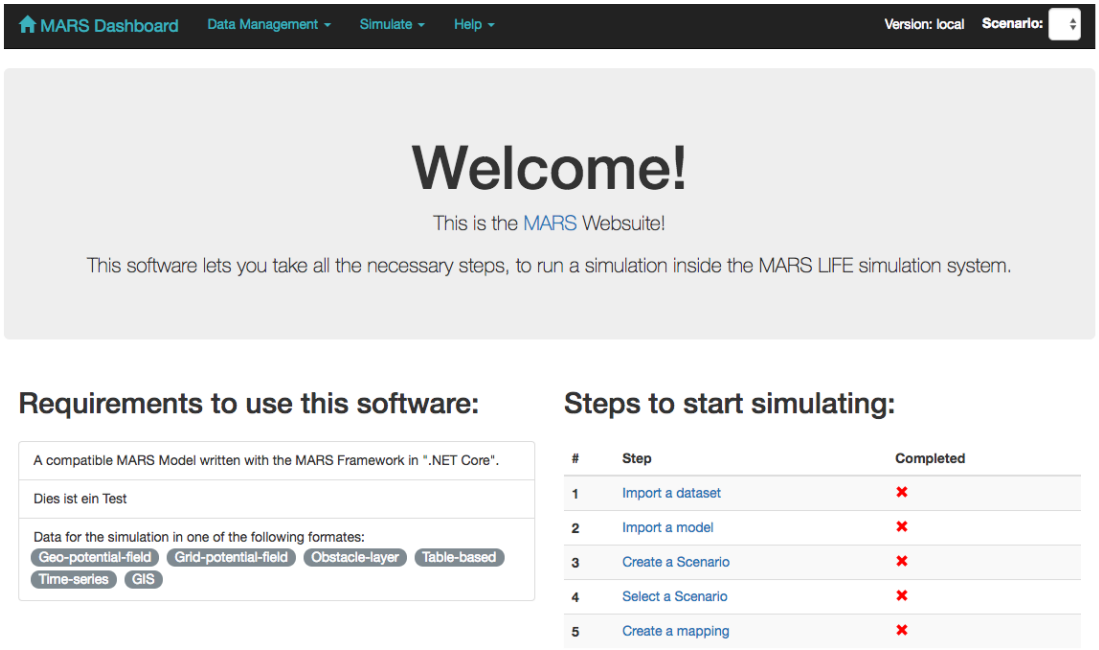

Abbildung 3.7: Startseite vom MARS Framework

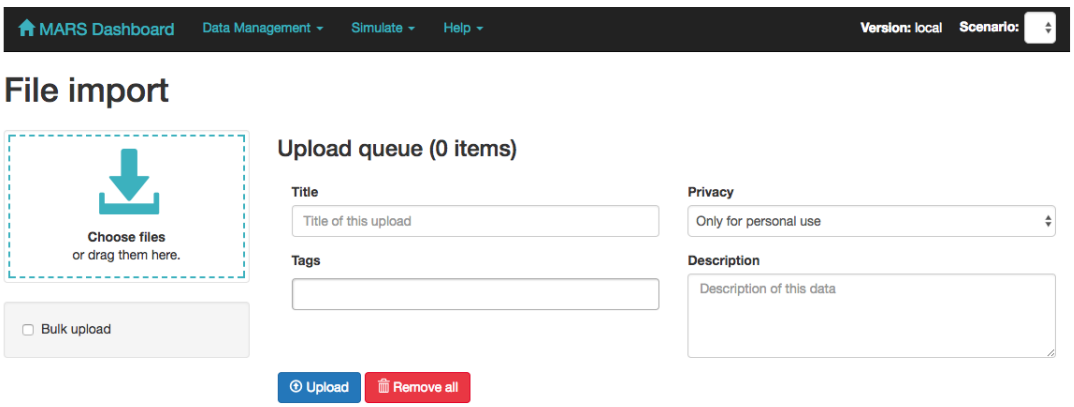

Abbildung 3.8: Seite vom MARS Framework für den Import von Dateien

Daher ist der Wissenstand über diesen Bereich sehr groß. Aus diesem Grund werden viele Entscheidungen für das Testkonzept für das MARS Framework auf Grundlage dieses Teilbereiches ausgeführt und somit auf die anderen Bereiche übertragen. Deswegen wird der Import-Bereich kurz vorgestellt, wie bereits im Hauptprojekt [Pie17a].

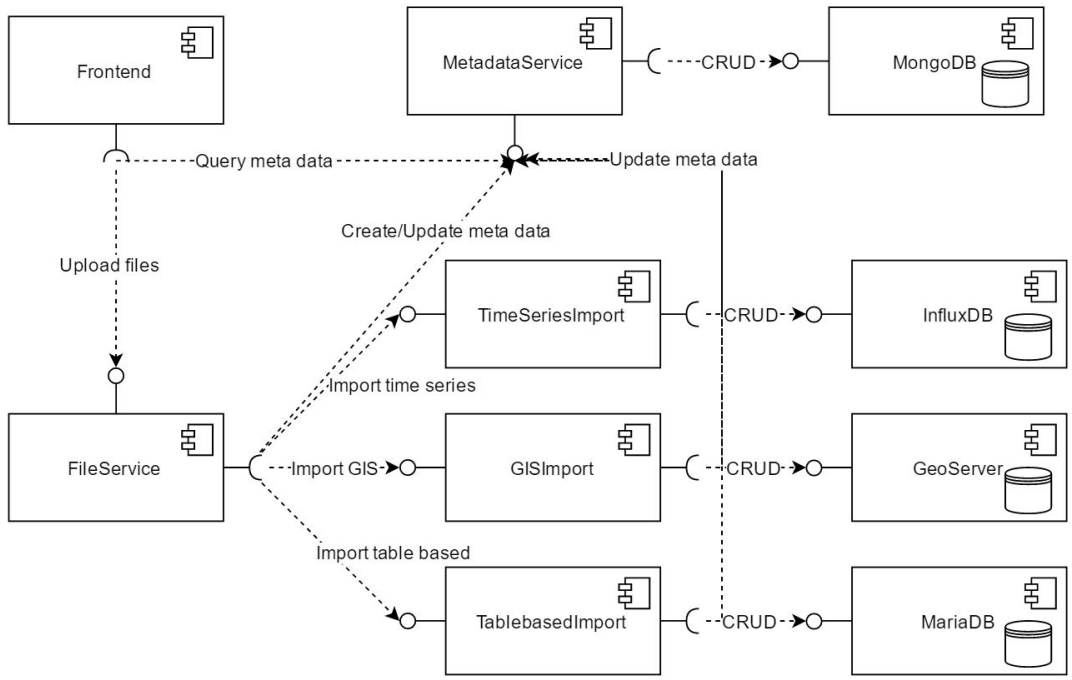

Abbildung 3.9: Diagramm der Komponenten für die Import Services [Quelle: MARS interne Dokumentation]

Der Import-Bereich im MARS Framework ist wie in Abbildung 3.9 aufgebaut. Das Frontend leitet alle seine Anfragen für das Importieren von Dateien weiter an den FileService. Dieser prüft, welche Datei in welchem Format importiert werden soll. Je nach Datei und Format leitet der FileService die Anfrage an einen der drei Microservices für den expliziten Import weiter. Davor erzeugt der FileService für die Datei beim Metadata-Service die entsprechenden Metadata. Diese Daten speichert der Metadata-Service in seine Datenbank ab.

Die drei Import-Services TimeSeriesImport, GISImport und TablebasedImport verhalten sich prinzipiell gleich. Alle importieren die Datei in ihre spezifische Datenbank und aktualisieren die Metadata zur Datei. Danach ist der Import abgeschlossen.

In der Abbildung 3.9 ist jedoch nicht dargestellt, dass für jede Kommunikation mit dem Metadata-Service das Objekt MetadataClient verwendet wird. Die Klasse MetadataClient

verwaltet die Kommunikation mit dem Service. Daher erstellen die anderen Services immer ein Objekt von dieser Klasse, um mit dem Metadata-Service zu kommunizieren.

4 Herausforderungen beim Testen von Microservices

In diesem Kapitel werden die Herausforderungen vorgestellt, die beim Testen von Microservice-basierten Systemen vorhanden sind. Diese Herausforderungen werden durch weitere ergänzt, die durch das MARS Framework selbst entstehen. Deren Darstellung folgt auf die Beschreibung der allgemeinen Herausforderungen.

Dazu werden im Folgenden verschiedene Testkonzepte vorgestellt und anschließend verglichen. Die meisten davon beschäftigen sich explizit mit Microservices. Dazu gehören die Testkonzepte von Wolff, Newman und Clemson. Das hier vorgestellte Konzept von Google bezieht sich im Gegensatz dazu nicht explizit auf Microservices.

Darüber hinaus wird in diesem Kapitel untersucht, inwieweit Exploratory Testing beim Testen von Microservice-basierte Systemen von Nutzen sein kann.

4.1 Herausforderungen beim Testen

In diesem Abschnitt werden die Herausforderungen beschrieben, die im Zusammenhang mit der Architektur Microservices stehen. Wichtig zu erwähnen ist, dass ausschließlich Herausforderungen thematisiert werden, die in Zusammenhang mit dem Testen stehen. Dadurch gibt dieser Abschnitt auch eine Übersicht darüber, an welchen Stellen sich das Testen von Microservices vom Testen einer monolithischen Anwendung unterscheidet.

Ergänzend dazu wurden die Herausforderungen beim Testen bereits in einer vorherigen Arbeit ausgearbeitet [Pie17b]. Die darin gewonnen Erkenntnisse fließen in diesen Abschnitt mit hinein. Zusätzlich werden in diesem Abschnitt Herausforderungen vorgestellt, die in der vorherigen Arbeit nicht Thema waren.

Die Architektur Microservices gleicht einer verteilten Anwendung, wegen der Aufteilung der Anwendung in mehrere eigenständige Services. Damit erbt die Architektur Microservices auch die Herausforderungen, welche im Zusammenhang mit dem Testen von verteilten Systemen stehen. Dies hat zur Folge, dass beim Testen verschiedene Aspekte beachtet werden müssen. Das Testen einer verteilten Anwendung unterscheidet sich vom Testen einer monolithischen

Anwendung, somit unterscheidet sich auch das Testen einer Anwendung mit Microservices von einer monolithischen Anwendung.

Da ausschließlich das Zusammenspiel der einzelnen Services die Gesamtfunktionalität ergeben, können die Abhängigkeiten unter den einzelnen Services zu Schwierigkeiten führen. Zum einem ist es für das Testen wichtig zu wissen, wie die einzelnen Services miteinander interagieren und voneinander abhängen. Dieses Wissen ist wichtig, um die Interaktionen bzw. die Abhängigkeiten zwischen den Services testen zu können.

Ergänzend dazu ist das Wissen notwendig, um die Services effizient und umfassend testen zu können. Damit dies gelingen kann, müssen die Abhängigkeiten isoliert werden. Andernfalls werden Abhängigkeiten mit getestet, was wiederum zu längeren Testlaufzeiten führen kann. Dadurch kann sich der Fokus der Tests verschieben, so dass die Abhängigkeiten getestet werden und nicht das definierte Ziel. Insbesondere können Fehler aus den Abhängigkeiten zu fehlgeschlagenen Tests führen, ohne dass das eigentliche Ziel getestet worden ist.

Das Ziel der Isolierung der einzelnen Tests kann zu verschiedenen weiteren Problemfeldern führen. Wie oben im Abschnitt zu den Besonderheiten beschrieben, sollte bei einem idealen Microservice-basierten System eine lose Kopplung zwischen den Services vorherrschen. Jedoch kann es aus unterschiedlichen Gründen, wie z.B. unglücklich definierten Aufgabenverteilungen, dazu kommen, dass keine lose Kopplung existiert, sondern die Abhängigkeiten unter den Services dadurch größer werden können. Je mehr Abhängigkeiten es gibt und besonders je unübersichtlicher die Abhängigkeiten sind, umso schwieriger wird es, die einzelnen Services zu isolieren.

Besonders das Testen der Schnittstellen bildet eine Herausforderung. Die Schnittstellen müssen umfassend getestet werden, weil sie die Grundlage für die Kommunikation zwischen den Services bilden. Dies kann sehr aufwendig sein, da die Schnittstellen viele verschiedene Pfade besitzen können. Erschwerend kommt hinzu, dass auch der Prozess der Isolierung sehr mühsam sein kann, weil die Schnittstellen viele verschiedene Abhängigkeiten besitzen können. Dahingehend muss der Aspekt der Effizienz Beachtung finden.

Die Aufgabe des Isolierens kann erschwert werden. Aufgrund der Aufgabenverteilung unter den einzelnen Services kann es Sinn ergeben, die Entwicklung der Services auf verschiedene Entwickler zu verteilen. Damit kann es jedoch schwierig werden, die Zusammenarbeit von Services zu testen. Es wird eine Zusammenarbeit und ein Wissenstransfer zwischen den verschiedenen Entwickler benötigt, um ein adäquates Testen der Services untereinander zu gewährleisten.

Je nachdem wie komplex ein Microservice ist, umso schwieriger kann es zu einem sein ihn zu isolieren. Zum anderen kann es auch eine Herausforderung sein, diesen Service für andere

Services zu simulieren. Damit die anderen Services den komplexen Service beim Testen nicht aufrufen, sondern eine für den Test erstellte Kopie.

Auch die Freiheit für jeden Microservice eine andere Technologie zu wählen, führt zu Herausforderungen. Bei der Wahl von verschiedenen Programmiersprachen müssen ggf. auch verschiedene Werkzeuge für das Testen eingesetzt werden. Dies erhöht den Aufwand von Schulungen und der Wartung der verschiedenen Werkzeuge.

Ein bei Microservices immer wieder erwähnter Vorteil, ist das schnelle und unabhängige Ändern der einzelnen Services. Dahingehend muss ein umfassendes Testkonzept Antworten darauf geben, wie es mit Änderungen umgeht. Besonders unter dem Aspekt, dass das Testen nicht das schnelle und unabhängige Ändern der Services verhindert.

4.1.1 MARS spezifische Herausforderungen

Ein Testkonzept für das MARS Framework muss neben den allgemeinen Herausforderungen auch die spezifischen Herausforderungen beantworten können. Daher ist es notwendig zu prüfen, ob beim MARS Framework spezifische Herausforderungen existieren und wenn ja, wie diese genau aussehen.

Für das MARS Framework existieren spezifische Herausforderungen, die zu einem aus der Struktur des Frameworks und dem aktuellen Stand des Projektes entstehen.

Das MARS Framework befindet sich in einem sehr fortgeschrittenen Zustand. Das heißt, dass das Framework kurz vor dem ersten Release steht. Zusätzlich finden am Framework regelmäßig auch gravierende Änderungen statt. Damit sind zu einem die Neuimplementierung von bestehenden Services und die Einführung von neuen Services gemeint. Dadurch können sich auch die Zuständigkeiten innerhalb des Systems verändern, also welcher Service welche Aufgaben übernimmt. Die erste Herausforderung besteht daher darin in ein bestehendes System ein Testkonzept zu entwerfen und zu etablieren.

Die zweite Herausforderung ist auf dem ersten Blick keine spezifische Herausforderung, weil auch andere Microservice-basierte Anwendungen ständig Änderungen erhalten. Diese Herausforderung wird dahingehend spezifisch, da das Testkonzept von einem externen Mitarbeiter entwickelt wird, der nicht zur MARS Gruppe gehört. Dadurch muss das Testkonzept flexibel auf Änderungen reagieren können, da ggf. nicht jede Änderung kommuniziert wird.

Eine weitere Herausforderung ist sozusagen ein Produkt aus den bisher genannten Herausforderungen. Durch den weiten Fortschritt des MARS Frameworks und die ständigen Änderungen auch an der Infrastruktur ist die Einarbeitung komplex. Besonders wegen den Änderungen an der Infrastruktur können erneute Einarbeitungen notwendig sein.

Ergänzend dazu wird die Einarbeitung durch veraltete oder nicht vorhandene Dokumentation erschwert. Besonders durch das Fehlen von definierten Anforderungen an das MARS Framework entsteht eine weitere Herausforderung. Das Fehlen von Anforderungen erschwert das Definieren von Testfällen enorm. Mit diesem Zustand muss das Testkonzept umgehen können.

Die nächste Herausforderung generiert sich aus der Struktur des Frameworks. Mit dem MARS Framework sollen Multi-Agenten Simulationen durchgeführt werden. Solche Simulationen können sehr aufwendig sein, besonders was Zeit und Ressourcen betrifft. Innerhalb des MARS Frameworks existieren Services, die nur bei der Simulation aufgerufen werden. Dabei stellt sich die Herausforderung, wie diese Microservices getestet werden können, ohne den Aufwand einer Simulation zu haben.

Erschwerend kommt hinzu, dass mit dem Framework verschiedene Simulationen ausgeführt werden können. Dadurch können sich auch die Anforderungen an das Gesamtsystem und die einzelnen Services verändern, die die Simulation durchführen. In diesem Bereich liegen generische Anforderungen vor, die erst durch die Simulation definiert werden können. Auf Grundlage dessen ist es schwierig für diesen Bereich Tests zu entwerfen, da die Anforderungen zu einem nicht vorliegen und zum anderem nicht eindeutig definiert werden kann, wie die Anforderungen aussehen, weil die Anforderungen in Abhängigkeit zur Simulation stehen.

Die nächste spezifische Herausforderung betrifft die Kapselung von Service-Aufrufen, um redundanten Sourcecode zu vermeiden. Beim Importieren von Dateien für die Simulationen werden für jede Datei sogenannte Metadata angelegt, die die importierten Dateien näher beschreiben. Da verschiedene Importarten existieren und dadurch auch mehrere Services, die Dateien importieren können, müssen alle Import-Services beim Metadata-Service die Erstellung der Metadata beauftragen. Damit nicht jeder Service diese Aufrufe und die dazugehörige Annahme der Antworten selber implementieren muss, wird dies im Objekt Metadataservice-Client gekapselt. Daher ergibt sich die Herausforderung, wie beim Testen, insbesondere auf der Ebene der Integrationstest, damit umgegangen werden soll. Zu einem können diese Aufrufe bei jedem Service, der sie verwendet, getestet werden. Dadurch können redundante Tests entstehen. Zum anderen können die Aufrufe nur bei einem Service mit getestet werden. Allerdings mit der Annahme, dass die Aufrufe, wenn sie bei einem Service funktionieren, auch bei allen anderen Services funktionieren. Auf dieses Problem muss im Testkonzept eine Antwort gefunden werden.

4.1.2 Umgang mit Änderungen

Neben den bisherigen Herausforderungen bilden vor allem der Umgang mit dem Änderungsprozess und den Änderung selbst eine Herausforderung. Ein großer Vorteil von Microservices ist das unabhängige Ändern der Microservices untereinander. Besonders durch die Modularisierung in kleinere Programme wird erwartet, dass bei Änderungen schneller eine neue Version bereitsteht als bei einem monolithischen System. Dahingehend muss das Testkonzept dabei unterstützen, dass der Prozess bei Änderungen effizient abläuft. Das Testen darf diesen Prozess nicht behindern.

Zusätzlich existieren verschiedene Arten von Änderungen. Mit diesen verschiedenen Arten muss das Testkonzept umgehen können. Eine Art von Änderung ist das Hinzufügen eines neuen Features. Eine andere Art ist das Ändern, um einen Fehler zu beheben. Eine Änderung kann zudem dadurch motiviert sein, dass eine Anforderung für eine Funktion sich verändert hat. Diese Änderung unterscheidet sich von den anderen beiden Arten.

Diese Änderungen sind nicht spezifisch für ein Microservice-basiertes System. Denn auch Anwendungen mit anderen Architekturen erhalten diese Arten von Änderungen.

Auf der Ebene der Architektur existieren spezifische Änderungen, die im Zusammenhang mit der Architektur Microservices stehen. Änderungen können darin bestehen, einen Service durch einen anderen Service zu ersetzen. Dazu kann es die Änderung geben, dass ein neuer Service dem System hinzugefügt wird. Dabei kann der Service Aufgaben von anderen Services übernehmen oder neue Funktionen hinzufügen. Des Weiteren kann auch ein Service nicht mehr benötigt werden und aus dem System entfernt werden. Dies ist auch eine eigene Art von Änderung.

Die verschiedenen Arten von Änderungen sind in der nachfolgenden Auflistung aufgelistet.

- Hinzufügen eines neuen Features
- Fehlerbehebung
- Änderung einer Anforderung
- Ersetzen eines Microservices
- Einführung eines neuen Microservices
- Wegfall eines Microservices

4.2 Aktuelle Testkonzepte zu Microservices

In diesem Abschnitt werden bestehende Testkonzepte zu Microservices vorgestellt. Das von Whittaker vorgestellte Testkonzept von Google ist nicht explizit für Microservices konzipiert. Es soll die Perspektiven der anderen Testkonzepte, die in dieser Arbeit vorgestellt werden, erweitern.

Innerhalb der Vorstellungen werden ausschließlich die Ansichten der jeweiligen Autoren wiedergegeben. Im Anschluss an die Vorstellungen werden die Konzepte miteinander verglichen. Dabei werden Gemeinsamkeiten und Unterschiede herausgearbeitet. Dazu werden die einzelnen Testkonzepte in Hinsicht auf ihre Stärken und Schwächen bewertet.

4.2.1 Testkonzept von Eberhard Wolff

In dem Buch “Microservices - Grundlagen flexibler Softwarearchitekturen" beschäftigt sich der Autor Eberhard Wolff mit der Architektur Microservices. Neben grundlegenden Informationen zu der Architektur und zum Aufbau einer Software mit dieser Architektur entwickelt Wolff auch Vorschläge für das Testen einer Anwendung, die Microservices verwendet. Sein Konzept zum Testen einer Anwendung mit Microservices wird in diesem Abschnitt vorgestellt [Wol15].

Das von Wolff entworfene Testkonzept unterscheidet zwischen dem Testen eines einzelnen Microservices und dem Testen des Gesamtsystems. Für Wolff ist besonders bei der Architektur Microservices wichtig, dass die Tests automatisiert ablaufen, denn nur damit kann ein unabhängiges und häufiges Deployment sicher gestellt werden. Andernfalls kann die notwendige Qualität für das Veröffentlichen gefährdet sein. Dazu empfiehlt Wolff, dass die Entwicklung der Microservices mit dem Test-Driven-Development(TDD) Ansatz erfolgen soll. Beim Test-Driven-Development Ansatz werden zuerst die Tests erstellt und im Anschluss die Funktion implementiert. Die Vorstellung seines Konzeptes beginnt von oben nach unten. Es wird zunächst das Testkonzept für das Gesamtsystem vorgestellt und im Anschluss das Konzept für einen einzelnen Microservice.

Testkonzept für das Gesamtsystem

Die Testpyramide in Abbildung 4.1 stellt das Testkonzept für das Gesamtsystem dar. Wolff versteht unter dem Gesamtsystem das System, welches aus dem Zusammenwirken der einzelnen Microservice entsteht. Die Entwicklung der Tests auf dieser Ebene sieht Wolff in der Verantwortung aller Teams, die einen Microservice für das Gesamtsystem entwickeln. Er begründet dies damit, dass durch das Testen des Gesamtsystems alle Microservices betroffen seien.

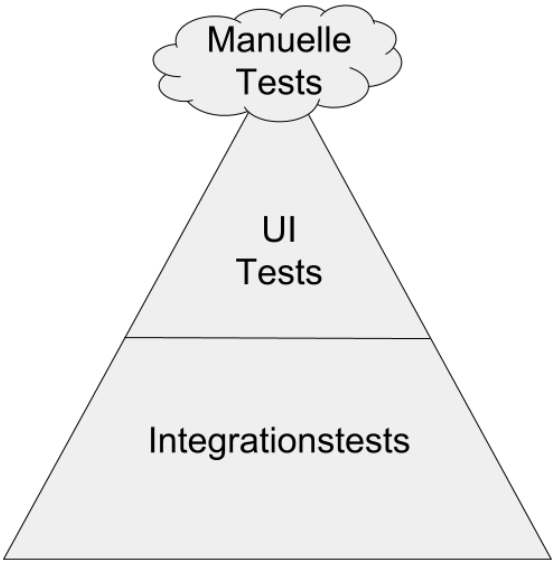

Abbildung 4.1: Testpyramide von Wolff für das Gesamtsystem [Wol15]

Auffallend in der Testpyramide in Abbildung 4.1 ist, dass nur zwei bzw. drei Ebenen existieren. Dies ergibt Sinn, wenn die dazugehörige Deployment Pipeline für das Gesamtsystem mit einbezogen wird. Diese ist in Abbildung 4.2 abgebildet.

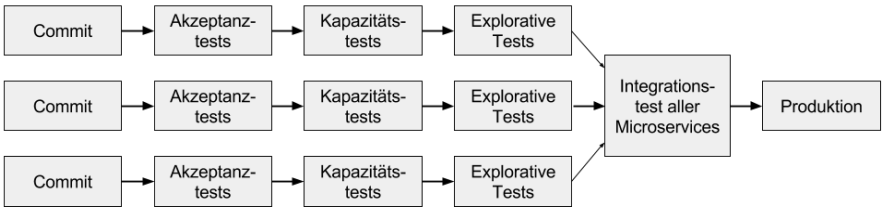

Abbildung 4.2: Deployment Pipeline von Wolff für das Gesamtsystem [Wol15]

Jeder Strang stellt einen Microservice mit seinen dazugehörigen Pipeline-Stationen dar. Nach Wolff soll jeder Microservice seine eigene Deployment Pipeline besitzen. In dieser Deployment Pipeline sollen die Tests für einen einzelnen Microservice durchgeführt werden, welche im nächsten Abschnitt beschrieben werden. Da sich die Testpyramide (Abbildung 4.1) ausschließlich auf das Gesamtsystem bezieht, entfällt die Ebene der Unit-Tests. Denn diese sind beim Testen eines einzelnen Microservices angelegt und sollen beim Zusammenführen der Microservices zum Gesamtsystem bereits durchgeführt worden sein.

Integrationstests

Laut Wolff ist das Ziel beim Testen des Gesamtsystems zu prüfen, ob beim Zusammenwirken der einzelnen Microservices Probleme entstehen. Da die Architektur Microservices eine Architektur der verteilten Systeme ist, existieren die gleichen Herausforderungen wie beim Testen von verteilten Systemen. Besonders durch das Zusammenwirken der einzelnen Microservices entsteht eine höhere Zahl von Fehlerquellen. Bei der Architektur Microservices wird dieses Problem mit dem Ansatz “Resilience" gelöst. Resilience bedeutet für Wolff in diesem Zusammenhang, dass die einzelnen Microservices auch dann funktionieren, wenn es Probleme mit anderen Microservices gibt.

Dafür stehen in der Testpyramide (Abbildung 4.1) und in der Pipeline (Abbildung 4.2) die Punkte Integrationstest und Integrationstest aller Microservices. Dabei ist es für Wolff wichtig, dass nur ein Service zur Zeit in diese Phase eintreten darf. Dadurch kann bei einem Fehler die Fehlerquelle leichter identifiziert werden. Wenn nur eine neue Version eines Microservices dazugekommen ist und nicht mehrere, kann die Verantwortung für den Fehler leichter zugeordnet werden.

Laut Wolff steckt jedoch hier ein großes Problem. Bei der Annahme, dass ein Integrationstest über alle Microservices eine Stunde dauert und nur ein Microservice in diese Phase eintreten darf, dann kann bei einem normalen Arbeitstag von acht Stunden nur achtmal pro Tag ein neuer Service veröffentlicht werden. Dies macht unter anderem einige Vorteile von Microservices, wie das unabhängige und schnelle deployen von Microservices, zunichte. Als Lösung schlägt Wolff vor, die Integrationstests von den einzelnen Microservices durchführen zu lassen und auf der Ebene des Gesamtsystems nur zu prüfen, ob der neue Microservice die anderen Microservices erreicht.

UI-Tests

Mithilfe von UI-Tests wird die Software über die Oberfläche getestet. Diese Tests sollen jedoch nur testen, ob die Oberfläche wie erwartet funktioniert. Indessen können die Tests fragil sein, da jede Änderungen an der Oberfläche dazu führen kann, dass der Test nicht mehr funktioniert. Ergänzend dazu sind die Tests häufig langsam, da das komplette System für die Tests benötigt wird.

Manuelle Tests

Zusätzlich zu den bisherigen Tests werden auch manuelle Tests durchgeführt. Mit manuellen Tests können z.B. neue Features getestet werden, die vielleicht noch nicht in der Produktion freigeschaltet sind.

Da manuelle Tests langwierig sein können, sollen Microservices auch veröffentlicht werden können, ohne vorher manuelle Tests durchgeführt zu haben. Dadurch wird die Veröffentlichung nicht verzögert. Die manuellen Tests können dann nach der Veröffentlichung durchgeführt werden.

Testkonzept für einen Microservice

Auch das Testkonzept für einen Microservice enthält verschiedene Arten von Tests. Dahingehend wird auch dieses Testkonzept in Form einer Pyramide dargestellt. Die dazugehörige Testpyramide ist in Abbildung 4.3 abgebildet. Zuständig für die Tests eines einzelnen Microservice ist für Wolff das Team, welches auch den Microservice entwickelt. Dazu gehört auch das Entwerfen und Erstellen der dazugehörigen Deployment Pipeline. Besonders wichtig beim Testen eines einzelnen Microservices ist, dass die Verbindung zu anderen Microservices simuliert wird. Zwar bestünde auch die Möglichkeit eine Referenzumgebung aufzubauen, in welcher aktuelle Versionen der anderen Microservices vorlägen, dies ist jedoch mit einem hohen Aufwand verbunden. Daher schlägt Wolff vor, die anderen Microservice zu simulieren, so dass sie auf Anfragen konstante Werte zurückgeben. Zeitgleich ist dieser Ansatz auch ressourcenschonender.

Um andere Microservices bzw. Abhängigkeiten zu simulieren, existieren laut Wolff zwei unterschiedliche Wege. Entweder werden die Abhängigkeiten durch einen Mock oder einen Stub simuliert. Wird ein Mock verwendet, dann wird ein Anruf mit einem vorher definierten Resultat simuliert. Damit besteht die Möglichkeit mehrere Aufrufe zu simulieren und zu überprüfen, ob die zu erwartenden Aufrufe stattgefunden haben. Ein Beispiel für ein Mock ist, dass für eine Kundennummer ein komplettes und vorher festgelegtes Objekt von einem Kunden zurück geliefert wird. Dadurch kann überprüft werden, ob das Objekt korrekt verarbeitet wird. Dazu kann mit einem Mock auch ein Fehler simuliert werden.

Mit einem Stub werden jedoch nur die Abhängigkeit und im begrenzten Umfang auch deren Funktionalitäten simuliert. Ein Beispiel dafür ist, dass der Stub auf Anfragen vorher definierte Antworten liefert. So können unter anderem Testobjekte mit bestimmten Eigenschaften definiert werden. Hiermit wird die Abhängigkeit simuliert.

Für Wolff sind Stubs die bessere Lösung, um Microservices zu simulieren, denn sie unterstützen nach seiner Aussage verschiedene Testszenarien. Daher kann ein Stub eines Microservices für alle abhängigen Microservices verwendet werden. Die Entwicklung eines Stubs liegt in der Verantwortung des Teams, welches den Microservice entwickelt. Dadurch soll sichergestellt werden, dass der Stub und der simulierte Microservice sich identisch verhalten. Notfalls muss dies mit eigenen Tests für den Stub sichergestellt werden. Von Vorteil ist, wenn alle Stubs mit derselben Technologie erstellt werden. Anderseits müssten zum Testen eines Microservices, welcher viele verschiedene andere Microservices nutzt, mehrere Technologien zum Erstellen eines Stubs verwendet werden.

Ergänzend dazu sollte die Technologie für die Stubs einen einfachen Technolgiestack nutzen. Dies ist mit dem Grund verknüpft, dass die Stubs weniger Ressourcen verbrauchen sollen als die eigentlichen Microservices.

In den folgenden Abschnitten werden die einzelnen Testarten näher vorgestellt.

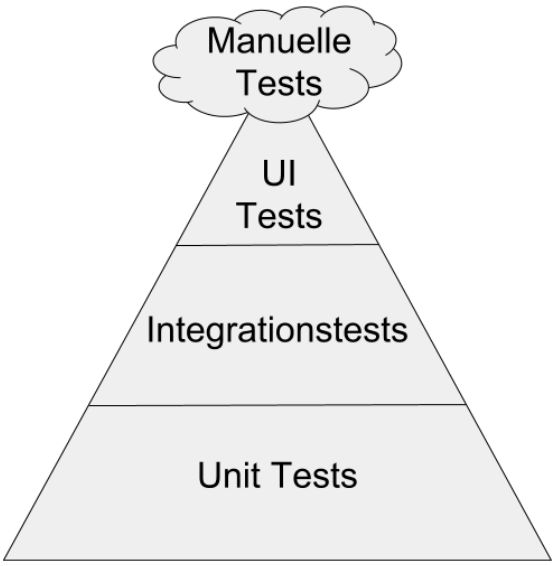

Abbildung 4.3: Testpyramide von Wolff für einen Microservice [Wol15]

Unit-Tests

Unter Unit-Tests versteht Wolff das Testen einzelner Bestandteile eines Systems und zwar der kleinsten möglichen Einheit. Damit ist das Testen von isolierten Funktionen bzw. Methoden gemeint. Für die Erstellung von Unit-Tests sind die Entwickler verantwortlich, da das Entwerfen

und Erstellen von Unit-Tests Wissen über die zu testende Funktion oder Methode voraussetzt. Der größte Vorteil von Unit-Tests ist die sehr geringe Ausführungsdauer. Dadurch können viele Unit-Tests in kurzer Zeit ausgeführt werden. Zusätzlich besteht die Möglichkeit jede Veränderung am Source-Code mit einem Unit-Test zu prüfen. Jedoch kommt die schnelle Ausführungsgeschwindigkeit nur zu Stande, wenn die Abhängigkeiten der Methoden und Funktionen voneinander isoliert werden. Andernfalls werden auch die Abhängigkeiten mit getestet. Die Isolierung kann darüber geschehen, indem die Abhängigkeiten simuliert werden. Hierbei kann wie beim Simulieren eines Microservice entweder ein Mock oder Stub genutzt werden. Jedoch lässt Wolff hier offen, welche Art von Simulation er bevorzugt.

Integrationstests

Bei den Integrationstests wird das Zusammenwirken des zu testenden Microservices mit den anderen Microservices getestet. Das Ziel dabei ist, Fehler bei der Integration der einzelnen Microservices zu vermeiden. Da die Verwendung von reellen Microservices sehr aufwendig ist, empfiehlt Wolff alle Microservices bis auf den, der getestet werden soll, zu simulieren. Wie oben schon beschrieben, kann dies mithilfe von Stubs umgesetzt werden. Die Tests können über das User Interface oder mit speziellen Testwerkzeugen erstellt werden. Eine Technologie benennt Wolf explizit und zwar den Ansatz des Consumer Driven Contract Tests. Für die Erstellung dieser Tests werde keine Kenntnises über das System benötigt.

UI Tests

Bei den UI Tests führt Wolff keine Unterscheidung zwischen den verschiedenen Ebenen auf, also zwischen dem Testen eines einzelnen Microservices und dem Testen des Gesamtsystems.

Manuelle Tests

Die Vorstellung von Wolff zu den manuellen Tests bei einem einzelnen Microservice sind sehr ähnlich wie beim Gesamtsystem. Jedoch kann es neben dem Testen eines neuen Features auch für weitere Aspekte verwendet werden, wie z.B. Sicherheit und Performance. Zum Testen, ob ein Fehler erneut auftaucht, sollen die manuellen Tests nicht verwendet werden. Solche Tests sollen automatisiert werden.

Last Tests

Last Tests werden zwar in seiner Testpyramide (Abbildung 4.3) nicht dargestellt, gehören aber zum Testkonzept für einen Microservice dazu. Wolff versteht unter Last Tests zwei verschiede-

ne Arten von Tests. Einmal den Performance-Test und zum anderen den Kapazitätentest. Bei dem Performance-Test wird geprüft, wie schnell die Anwendung reagiert, und bei dem Kapazitätentest, wie viele Anfragen das System zeitgleich bearbeiten kann. Damit untersuchen beide Tests die Leistungsfähigkeit der Anwendung. Zusätzlich kann auch der Ressourcenverbrauch untersucht werden.

Ergänzende Möglichkeiten

Für Wolff reichen Tests alleine nicht aus, um Fehler zu finden. Daher sollen Logging und Monitoring dabei unterstützen Fehler zu finden. Besonders bei Microservices, welche schon in Produktion sind, ist dies sehr hilfreich.

Logging und Monitoring

Durch das Logging der Microservices können Fehler gefunden werden, die dem Endanwender nicht auffallen. Jedoch ist es dabei wichtig, dass die Einträge im Log automatisch ausgewertet werden. Hiermit bekommt das verantwortliche Team auch Kenntnis darüber, dass ein Fehler aufgetreten ist. Zusätzlich unterstützen die Log-Dateien bei der Suche nach Fehlern. Allerdings muss das Logging auf die Architektur Microservices angepasst werden.

Wenn jeder Microservice nur sein Verhalten loggt, kann es dazu führen, dass es nicht nachvollziehbar ist, wie der genaue Ablauf war. Ergänzend dazu kommt, dass das Schreiben in eine Log-Datei unzureichend sein kann, wenn es mehrere Instanzen eines Microservices gibt oder eine neue Version eines Microservices deployt wird. Beides kann dazu führen, dass sich die Log-Einträge auf mehrere Dateien verteilen oder unvollständig sind. Daher schlägt Wolff vor, dass die Logs zentral verwaltet werden. Es soll einen zentralen Log-Server geben, an den die Log-Einträge geschickt werden. Dieser wertet die empfangenen Log-Einträge aus und kann dabei wichtige Informationen herausfiltern. Der größte Vorteil für Wolff ist, dass die Log-Einträge zentral gespeichert werden und damit die Verbindungen zwischen einzelnen Aktionen nachvollziehbar werden.

Des Weiteren empfiehlt Wolff die Microservices via Monitoring zu überwachen. Im Gegensatz zum Loggen nutzt Monitoring bestimmte Metriken, um einen Microservice zu überwachen. Diese Metriken können z.B. CPU-Auslastung und RAM-Auslastung sein. Auf Grundlage dieser Metriken können Fehler gefunden werden. Ein Beispiel dafür ist, dass durch eine sehr hohe RAM-Auslastung ein Fehler gefunden wird, der zu einem Memory Leak geführt hat.

4.2.2 Testkonzept von Sam Newman

In dem Buch “Building Microservices" von Sam Newman wird neben der generellen Architektur Microservices und dem Erstellen von Microservices auch das Thema Testen genauer betrachtet. Das von Newman im Buch präsentierte Testkonzept wird in diesem Abschnitt vorgestellt.

Auch das Testkonzept von Newman basiert auf einer Testpyramide, welche in Abbildung 4.4 dargestellt ist. Dabei muss erwähnt werden, dass seine Testpyramide auf der Testpyramide von Mike Cohn basiert, welche im Buch “Succeeding With Agile" vorgestellt wird [Coh09].

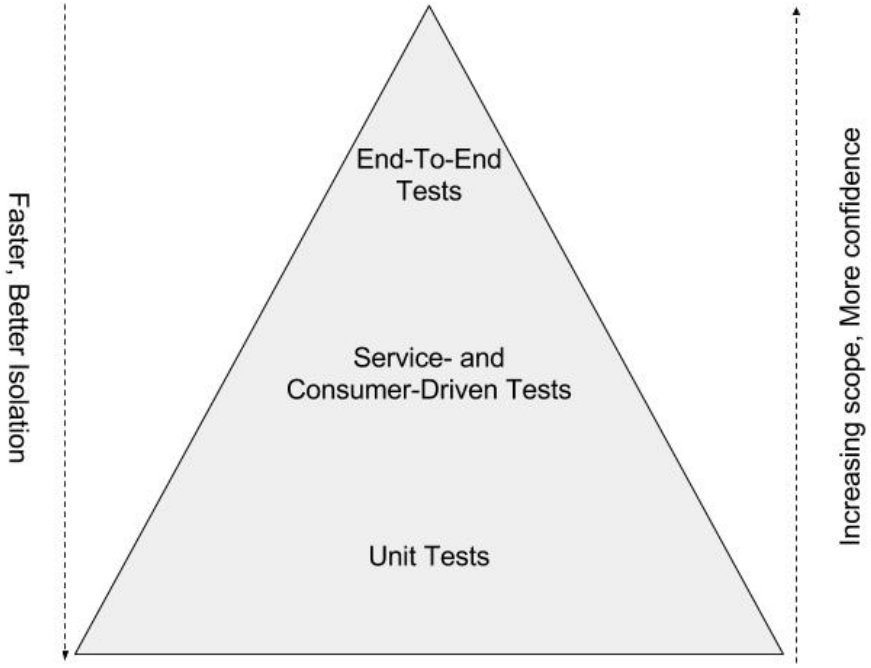

Abbildung 4.4: Testpyramide von Newman [New16]

Die Testpyramide von Newman unterscheidet sich im Groben in zwei Punkten von der Testpyramide von Cohn. Zum einem heißt die oberstes Ebene bei Cohn nicht End-To-End Tests, sondern UI-Tests, und zum anderen bezeichnet Cohn die mittlere Ebene nur mit Service. Sie enthält nicht den expliziten Verweis auf Consumer Driven Contract Tests.

Allerdings trifft diese Testpyramide Aussagen über den Testumfang und dem damit verbundenen Vertrauen in die Anwendung sowie in die Ausführungsgeschwindigkeit und der Isolierung des jeweiligen Tests. Je weiter oben sich der Test in der Pyramide befindet, umso höher ist die Ausführungsdauer und umso geringer isoliert ist der Tests. Jedoch steigen der Testumfang und das Vertrauen in die Anwendung.

Bei der Vorstellung des Konzeptes von Newman wird mit den End-To-End Tests begonnen, da die Definition dieser Testebene Einfluss auf die anderen Testebenen besitzt.

End-To-End Tests

Für Newman testen End-To-End Tests das gesamte System. Dabei laufen diese Tests häufig über die Benutzeroberfläche wie z.B. GUI und/oder Weboberfläche ab. Jedoch können sie auch über eine Schnittstelle des Systems ablaufen. Eine Einschränkung auf GUI oder Schnittstelle existiert nicht. Im Detail bedeutet dies, dass über Interaktionen mit der Benutzeroberfläche oder einer Schnittstelle das gesamte System getestet werden soll. Dabei können Interaktionen, wie das Hochladen einer Datei getestet werden.

Die End-To-End Tests geben ein gutes Gefühl, wenn sie erfolgreich ablaufen. Der Grund dafür ist, dass sie einen großen Teil der Anwendung und somit auch vom Sourcecode abdecken. Dies zeigt auch die Testpyramide von Newman in der Abbildung 4.4. Wie der Abbildung entnommen werden kann, erhöht sich das Vertrauen in die Software je weiter oben sich die Testart befindet. Da sich die End-To-End Tests ganz oben befinden, sorgen sie auch für ein hohes Vertrauen in die Software.

Trotz des hohen Vertrauens, die End-To-End Tests geben, rät Newman dazu, auf End-To-End Tests zu verzichten. Für die Ablehnung führt er mehrere Gründe an.

Wie der Abbildung 4.4 entnommen werden kann, ist die Isolierung bei End-To-End Tests am geringsten bzw. es existiert keine. Dies liegt daran, dass beim End-To-End Test keine Services simuliert werden sollen. Daher müssen alle Services für den Test veröffentlicht und gestartet werden. Dies erhöht den Aufwand für die Durchführung der Tests und den Umfang für die Tests.

Der größere Testumfang bringt weitere Probleme mit sich. Newman gibt dazu zu bedenken: “As test scope increase, so too do the number of moving parts" [New16]. Je mehr bewegliche Teile es gibt, umso zerbrechlicher werden die Tests. So könnte ein temporärer Fehler im Netzwerk für das Scheitern eines Tests sorgen, ohne eine Aussage über die zu testende Funktionalität zu geben. Das kann dazu führen, dass die Tests weniger deterministisch ablaufen.

Nicht-deterministische Tests sind für Newman ein Problem. Solche End-To-End Tests die mal erfolgreich ablaufen und mal nicht, obwohl keine Änderung vorliegt, werden auch “flaky tests" [New16] genannt. Diese Tests haben häufig die Folge, dass nach einem Fehlschlag der Test einfach wiederholt wird. Ist der Test dann erfolgreich, wird das vorherige Scheitern nicht beachtet.

Nach Newman kann dadurch das Vertrauen in den Testfall verloren gehen. Zusätzlich besteht die Gefahr, dass das Scheitern des Tests nicht als Problem, sondern als der Normalfall angesehen

wird. Damit wird der Testfall ein Opfer des sogenannten “normalization of deviance" [Vau97] werden.

Die nicht vorhandene Isolierung und der größere Testumfang haben auch Auswirkungen auf die Dauer der Tests. Dies kann auch der Pyramide (Abbildung 4.6) entnommen werden. Denn je weiter oben sich eine Testart befindet, umso geringer ist die Ausführungsgeschwindigkeit.

Die Dauer der Tests erhöht sich dadurch, dass reelle Services verwendet werden. Deren Performance ist meist schlechter als ihre simulierten Kopien, die explizit für das Testen angelegt werden. Diese Kopien liefern jedoch nur festgelegte Antworten auf definierte Anfragen.

Die hohe Dauer bildet für Newman einen weiteren Grund für die Ablehnung von End-To-End Tests. Denn End-To-End Tests können mehrere Stunden dauern. Besonders im Zusammenspiel mit “flaky tests" ist dies für Newman ein sehr großes Problem. Wahrscheinlich wird der Test erneut gestartet, wie oben bereits ausgeführt. Dadurch wird die doppelte Zeit investiert.

Selbst wenn die Tests stabil ablaufen, kann die Testdauer zum Problem werden. Die hohe Testdauer kann dazu führen, dass zu viel Zeit vergeht bis festgestellt wird, dass eine Anforderung nicht korrekt funktioniert. Dies kann zur Folge haben dass die zuständigen Entwickler gedanklich inzwischen woanders sind und sich dann erst wieder in die Tests einarbeiten müssen.

Jedoch können für Newman End-To-End Tests übergangsweise verwendet werden, solange der adäquate Ersatz nicht geschaffen worden ist. Für diese Situation empfiehlt Newman die Anzahl von End-To-End Tests zu reduzieren. Mit der Reduzierung sinkt zu einem die Gesamtdauer für die End-To-End Tests und zum anderen müssen weniger Tests gewartet werden. Als Beispiel nennt er eine sehr niedrige zweistellige Anzahl für ein komplexes System.

Eine Fokussierung auf Kernfunktionen kann dabei unterstützen End-To-End Tests zu verringern. Für ihn ist der Ansatz, dass jede User Story mit einem Testfall auf dieser Ebene abgedeckt werden soll, sehr ungeeignet. Durch dieses Verfahren entstehen für ihn zu viele Tests.

Ergänzend dazu kann auch der Umgang mit instabilen Tests dabei unterstützen Tests zu reduzieren. Dafür zieht Newman eine Ausarbeitung von Martin Fowler hinzu [Fow11]. Grundsätzlich sollen instabile Tests entfernt werden. Allerdings kann zuerst versucht werden, die Tests zu reparieren, damit sie deterministisch ablaufen. Alternativ können die Tests auch komplett neu geschrieben werden. Beides geschieht unter der Voraussetzung, dass der Test erhalten bleiben soll.

Alle Funktionen, die nicht bzw. nicht mehr über End-To-End Tests abgedeckt werden, müssen über andere Tests auf den anderen Ebene abgedeckt werden.

Existieren End-To-End Tests, liegt für Newman ein weiteres Problem vor und zwar in der Deployment Pipeline. Bei einem für Newman naiven Ansatz, der in Abbildung 4.5 dargestellt

ist, sind die einzelnen Schritte einfach aneinander gehängt. Bei diesem Ansatz hätte jeder Service eine eigene Deployment Pipeline. Das bedeutet auch, dass jeder Service seine eigene Ebene für End-To-End Tests besitzt. Hieraus entstehen verschiedene Fragen bzw. Probleme.

Es stellt sich die Frage, welche Versionen von den anderen Services verwendet werden sollen. So können die Versionen verwendet werden, welche aktuell in Produktion sind oder die aktuellsten Entwicklungsversionen. Diese Frage ist für Newman nicht leicht zu beantworten.

Abbildung 4.5: Pipeline von Newman mit eigener Ebene für End-To-End Tests [New16]

Ein weiteres Problem entsteht in der Testumgebung. Wenn jeder Microservice seine eigene Ebene für End-To-End Tests hat, muss auch für jeden Microservice eine eigene Testumgebung vorhanden sein. Dies kann zu unnötigen Duplikaten führen und somit auch zur unnötiger Arbeit.

Die Lösung für Newman ist, dass es für alle Microservices nur eine Umgebung der End-To-End Tests vorhanden ist. Bei diesem Konzept wird jeder Service nach dem erfolgreichen Abschließen seiner Deployment Pipeline in diesen Schritt überführt, wie in Abbildung 4.6 dargestellt. Dabei kann immer nur ein Microservice zurzeit diese Ebene betreten. Ansonsten wird die Suche nach Fehlerquellen erschwert, da nicht ein Service in einer neueren Version vorliegt, sondern zwei oder ggf. mehrere. Dadurch existieren mehrere mögliche Orte für die Fehlerquelle.

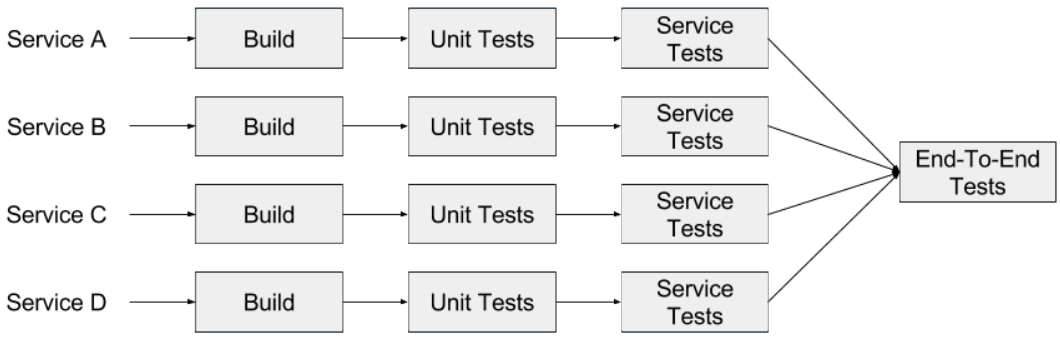

Abbildung 4.6: Pipeline von Newman mit einer End-To-End Tests Ebene für alle Microservices [New16]

Jedoch enthält auch dieses Konzept Probleme. Da es nur eine Ebene für End-To-End Tests gibt, kann auch nur ein Microservice zurzeit getestet werden. Dies hat Auswirkungen darauf, wie viele Services an einem Tag veröffentlicht werden können.

Neue Microservices werden nur veröffentlicht, wenn sie die gesamte Deployment Pipeline erfolgreich abgeschlossen haben. Deswegen müssen sie ggf. warten, bis sie die Ebene der End-To-End Tests betreten können. Dieses Problem verschlimmert sich, je länger die End-To-End Tests dauern.

Besonders problematisch wird es, wenn die End-To-End Tests fehlerhaft sind und repariert werden müssen. In dieser Zwischenzeit können keine neuen Tests gestartet werden, da die Tests zuerst repariert werden müssen. Jedoch kann auch der Entwicklungsprozess nicht einfach gestoppt werden. Dies kann dazu führen, dass mehrere Microservices sozusagen im Stau stehen, um die End-To-End Tests durchführen zu können. In der Zwischenzeit kann kein neuer Microservice veröffentlicht werden.

Service- and Consumer-Driven Tests

Im Umfeld der Architektur Microservices versteht Newman die Service Tests als Testen eines einzelnen Services. Dabei wird der Microservice mit seinen Funktionen direkt getestet. Der Grund für das Testen eines einzelnen Service ist die Eingrenzung des Testumfangs.

Der unter Test stehende Microservice muss isoliert werden, damit ausschließlich dieser Microservice getestet wird. Dadurch wird die Dauer der Tests geringer und der Testumfang wird weiter eingegrenzt.

Um einen Service zu isolieren, müssen die externen Abhängigkeiten simuliert werden. Externe Abhängigkeiten können dabei andere Services oder Datenbanken sein. Für das Simulieren externer Abhängigkeiten existieren für Newman zwei unterschiedliche Möglichkeiten. Eine Möglichkeit für ihn sind Stubs und die andere sind Mocks.

Für Newman simuliert ein Stub Antworten auf Anfragen. Ein Stub kann dahingehend erstellt werden, dass er auf bestimmte Anfragen vorher definierte Antworten liefert. So kann der Stub auf eine bestimmte Kundennummer einen vorher definierten Wert zurück liefern. Jedoch kann mit einem Stub nicht überprüft werden, wie oft er oder eine Methode von ihm aufgerufen wurde.

Dies ist bei einem Mock anders. Ein Mock bietet mehr Funktionen als ein Stub. Es kann überprüft werden, ob ein Aufruf stattgefunden hat. Zusätzlich können auch die Anzahl der Aufrufe und mögliche Nebeneffekte mit in die Überprüfung einfließen. Dadurch besteht die Möglichkeit mit einem Mock zu testen, ob ein Aufruf einer bestimmten Funktion z.B. nur

einmal passiert oder öfter. Ein zu prüfender Nebeneffekt ist z.B., dass beim Anlegen eines neuen Nutzers auch automatisch das Anlegen einer Bankverbindung geschieht.

Für die Implementierung eines Mock wird mehr detailliertes Wissen benötigt als bei der Implementierung eines Stubs. Dazu ist auch die Implementierung eines Mocks aufwendiger.

Die Balance zwischen der Verwendung von Stubs und Mocks ist für Newman äußerst labil und risikobehaftet. Für ihn ist es nicht vom Vorteil nur eine dieser Methoden zu verwenden. Dies trifft auch für Unit-Tests zu.

Persönlich favorisiert Newman die Verwendung von Stubs. Jedoch ist ein Tool, welches sowohl Stubs als auch Mocks anbietet, von großem Vorteil.

Aufgrund der Isolierung können Service Tests schneller als Unit-Tests sein. Dies ist abhängig davon, wie die einzelnen Tests entworfen worden sind.

Trotz der Isolierung der Service Tests kann es Sinn ergeben, externe Komponenten wie Datenbanken nicht zu simulieren. Dadurch erhöht sich wiederum die Dauer der Tests.

Wie oben bereits erwähnt, wird der Testumfang aufgrund der Isolierung geringer. Der Testumfang ist jedoch weiterhin größer als bei den Unit-Tests. Das hat zur Folge, dass ggf. die Suche nach der Fehlerquelle erschwert wird im Vergleich zu den Unit-Tests. Dennoch erleichtert die Isolierung die Suche nach der Fehlerquelle, da nur ein Microservice getestet wird.

Zusätzlich zu den Service Tests empfiehlt Newman Consumer Driven Contract Tests. Seine Empfehlung beruht dabei hauptsächlich auf den Nachteilen, die er bei End-To-End Tests sieht, welche auch im vorherigen Abschnitt beschrieben worden sind.

Unit Tests

Auf dieser Ebene werden explizit eine Funktion bzw. Methode getestet. Es werden keine Services gestartet und keine externen Ressourcen verwendet. Dabei stehen ausschließlich die technischen Aspekte im Fokus.

Das Hauptziel für Newman ist, dass die Unit-Tests sehr schnelles Feedback darüber geben, ob die jeweiligen Methoden bzw. Funktionen sich wie erwartet verhalten. Mit diesem schnellen Feedback möchte er die meisten Fehler finden.

Für Newman ist das Ziel erstrebenswert in einer Sekunde mehrere Tausend Unit-Tests auszuführen. Um dieses Ziel zu erreichen, müssen die Unit-Tests eine sehr geringe Dauer haben.

Die geringe Dauer wird über einen hohen Grad an Isolierung erreicht. Dafür werden alle externen Abhängigkeiten simuliert. Wie bereits bei den Service Tests beschrieben, können dafür Stubs oder Mocks verwendet werden.

Die Unit-Tests besitzen von allen Testarten den höchsten Grand an Isolierung. Dies ist anhand der Position der Unit-Tests in der Pyramide (Abbildung 4.4) erkennbar.

Die Unit-Tests bilden die Grundlage für jede Änderung am Sourcecode. Denn mit geringen Aufwand wird festgestellt, ob eine Änderung Fehler verursacht.

Testen nach der Produktion

Für Newman ist es wichtig, dass auch nach der Produktivsetzung einzelner Services bzw. des Gesamtsystems weiterhin getestet wird. Im produktiven Betrieb werden immer wieder Fehler auftauchen. Die Lösung, das erneute Auftreten durch Tests vor der Produktivsetzung abzusichern, stößt an einem bestimmten Punkt an seine Grenzen. Dahingehend schlägt Newman zwei verschiedene Deployment-Strategien für das Veröffentlichen eines neuen Services vor.

Eine Strategie ist das sogenannte “blue/green deployment" [New16]. Bei dieser Strategie wird parallel zu dem Service, welcher sich in Produktion befindet, die neue Version veröffentlicht. Die Anfragen werden zu diesem Zeitpunkt weiterhin an die alte Version weitergeleitet.

Nachdem die neue Version veröffentlicht wurde, kann getestet werden, ob das Deployen korrekt funktioniert hat. Dies kann mit sogenannten “smoke tests" [New16] umgesetzt werden. Die Smoke Tests haben nur die Aufgabe zu testen, ob das Deployen funktioniert hat.

Bei einer erfolgreichen Bestätigung des Deployments können die Anfragen an den neuen Servicesumgeleitet werden. Sollten in der Produktion Fehler auftauchen oder der neue Service ausfallen, können die Anfragen wieder an die vorherige Version umgeleitet werden.

Das Wechseln zwischen den einzelnen Versionen kann für den Nutzer völlig unsichtbar ablaufen. Damit bietet diese Strategie ein zero-downtime Deployment an.

Die andere Strategie, welche Newman vorstellt, ist das sogenannte “canary releasing" [New16]. Auch bei dieser Strategie wird die neue Version eines Services parallel zur bisherigen Version veröffentlicht. Jedoch wird hier nicht auf einmal auf den neuen Service umgestellt, sondern Stück für Stück. Das heißt, dass ein Teil der ankommenden Anfragen an den neuen Service weitergeleitet werden, während der Rest an die bisherige Version geht. Diesen Ansatz verwendet unter anderem Netflix für das Veröffentlichen neuer Versionen [New16].

Mit der Umleitung von produktiven Anfragen an den neuen Service können bei diesem funktionale und nicht-funktionale Anforderungen getestet werden. Sollte der Service diese nicht erfüllen, wären zum einen nur ein kleiner Teil der Anfragen und somit auch der Nutzer betroffen und zum anderen kann wieder schnell auf die bisherige Version um gestellt werden.

Alternativ zum bisherigen Ablauf können auch die kompletten Anfragen kopiert werden und an den neuen Service weitergeleitet werden. Im Anschluss können dann die Antworten vom bisherigen Service und dem neuen Service verglichen werden, sowohl funktional als auch

nicht-funktional. Sollten dabei keine Probleme auftauchen, werden die Antworten vom neuen Service weitergegeben. Bis dahin werden ausschließlich die Antworten vom bisherigen Service weiterverwendet und die vom neuen Service werden nach dem Vergleichen einfach ignoriert.

Im Vergleich zum blue/green depolyment ist canary releasing komplexer einzurichten und zu verwalten. Jedoch existieren beide Versionen länger nebeneinander und der Anteil der Anfragen kann leichter verteilt werden.

Neben den bisher beschriebenen funktionalen Tests schlägt Newman auch die Verwendung von sogenanntem Cross-Functional Testing vor. Cross-Functional Testing bedeutet für ihn die Mischung von funktionalen und nicht-funktionalen Tests, da sie für ihn nicht immer unterscheidbar sind. Ein Beispiel für ihn sind Perfomance Tests.

Die Tests im Umfeld von Cross-Functional Testing sollen ebenso wie die funktionalen Tests in Form einer Pyramide angeordnet werden. Dabei müssen manche Tests wie End-To-End Tests sein, wie z.B. Load Tests. Eine Pyramide wie in Abbildung 4.4 stellt Newman hier nicht bereit.

Es kann, wie bei den funktionalen End-To-End Tests, Sinn ergeben, diesen Test durch eine andere Testart mit kleinerem Testumfang zu ersetzen. Als Beispiel führt Newman an, dass beim Feststellen eines Flaschenhalses während eines End-To-End Performance Test dieser durch einen Performance Test auf einer unteren Testebene abgedeckt werden soll. Dieser soll dann in Zukunft die Ursache für den Flaschenhals prüfen.

Newman empfindet die Performance Tests besonders bei einer Anwendung mit der Architektur Microservices sehr wichtig, da der Netzwerkverkehr deutlich höher ist als bei einer monolithischen Anwendung. Dadurch kann die Performance der Gesamtanwendung abnehmen. Unter anderem auch deshalb rät er dazu, Performance Tests möglichst frühzeitig einzusetzen, damit die Performance der Anwendung und der einzelnen Microservices ohne unnötigen Zeitverlust getestet werden können.

Wichtig ist dabei, dass die Tests den Anforderungen in der Produktion so nah wie möglich kommen. Daher muss bei den Performance Tests eine Grundlast produziert werden, welche der in der Produktion ähnlich ist. Nur dann sind die Tests auch aussagekräftig. Das heißt, dass mehrere Consumer mit ihren Anfragen simuliert werden müssen. Zusätzlich funktionieren diese Tests nur, wenn sie mit einem Monitoring der Anwendung zusammenarbeiten.

Ergänzend dazu müssen für die Tests eindeutige und messbare Ziele definiert werden. Andernfalls ist es nicht möglich zu entscheiden, wann der Test erfolgreich war und wann nicht. Deswegen muss definiert sein, wie lange ein Test höchstens dauern darf.

Im Gegensatz zu den funktionalen Tests sollen die Performance Tests nicht bei jedem Commit ausgeführt werden, da es laut Newman nicht immer möglich ist. Allenfalls eine Untermenge

soll pro Tag ausgeführt werden und einmal in der Woche kann eine größere Menge ausgeführt werden. Jedoch sollten die Abstände zwischen den Ausführungen von Performance Tests nicht zu groß sein. Ansonsten wird es schwieriger, die Änderung zu finden, welche für die schlechtere Performance verantwortlich ist.

4.2.3 Testkonzept von Toby Clemson

Das von Toby Clemson entworfene Testkonzept [Cle14] wurde bereits in der Ausführung zum Hauptseminar vorgestellt [Pie16]. In der Ausarbeitung wurde das Testkonzept jedoch nur sehr knapp dargestellt. Daher wird es hier ausführlicher behandelt.

Für Clemson existiert ein klares Schema, wie jeder Microservice grundsätzlich aufgebaut ist. Dieses Schema ist in Abbildung 4.7 zu sehen. Auf der Grundlage dieses Schemas entwickelte Clemson sein Testkonzept. Die einzelnen Schichten werden von ihm nicht näher erläutert.

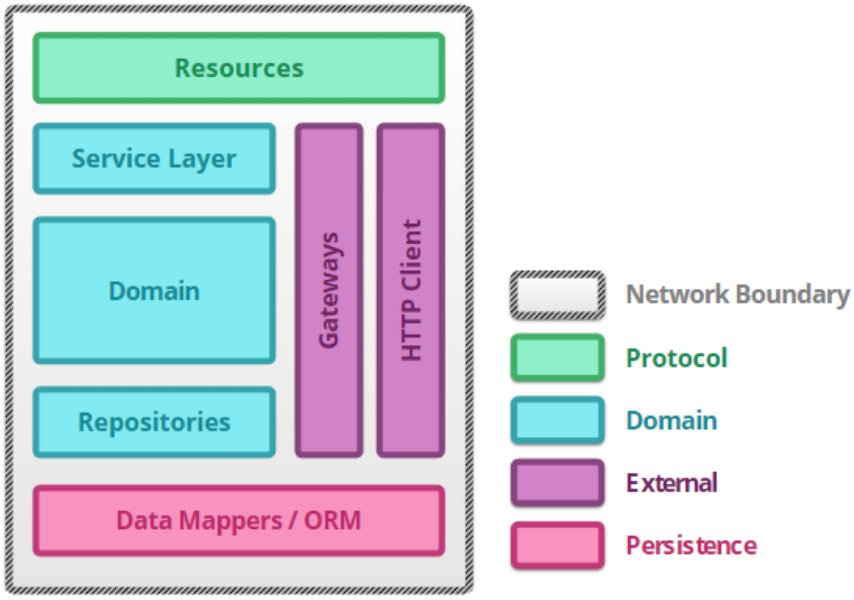

Abbildung 4.7: Schema der Microservices von Clemson [Cle14]

Auch das Testkonzept von Clemson wird mit einer Testpyramide dargestellt, wobei er sich indirekt auf die Testpyramide von Mike Cohn bezieht [Fow12, Coh09]. Die von Clemson entworfene Testpyramide ist in Abbildung 4.8 abgebildet.

Die Position innerhalb der Testpyramide gibt auch hier wieder, wie viele Tests von diesen Testarten in Relation zu den anderen Testarten durchgeführt werden sollen. Zusätzlich wird

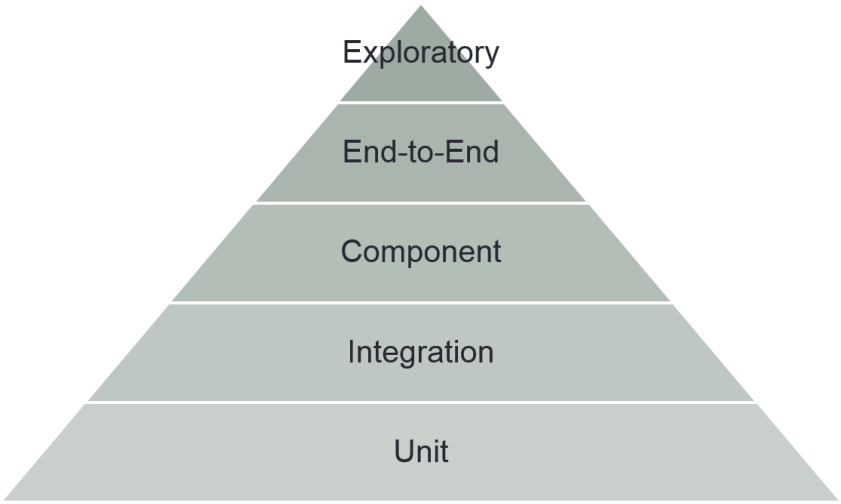

Abbildung 4.8: Testpyramide von Clemson [Cle14]

auch der Umfang wiedergegeben. Je weiter oben, umso größer ist auch der Bereich, der durch die Tests abgedeckt wird.

Ein Vorteil bei einer Microservices Anwendung ist, dass für jeden Service einzeln entschieden werden kann, wie detailliert er getestet werden soll. Enthält der Service sehr wichtige Geschäftslogik kann es Sinn ergeben, diesen Service ausführlicher zu testen als einen anderen Service.

Die einzelnen Testarten werden in den nächsten Abschnitten näher erläutert. Begonnen wird mit den Unit-Tests und danach geht es in der Pyramide weiter nach oben.

Unit-Testing

Für Clemson wird beim Unit-Testing die am kleinsten zu testende Einheit einer Software getestet. Es geht dabei darum, ob die zu testende Einheit das erwartende Verhalten wiedergibt.

Die Größe der zu testenden Einheit ist dabei nicht eindeutig definiert. Jedoch finden die Tests häufig auf der Ebene der Klassen statt oder es wird eine kleine Gruppe von verbundenen Klassen getestet. Je kleiner der Testumfang ist, desto leichter kann das Verhalten gegenüber dem zu erwartenden Verhalten getestet werden.

Zusätzlich können Unit-Tests sehr hilfreich beim Designen der Software sein. Sie unterstützen dabei Module in unabhängige Einheit zu unterteilen, damit diese isoliert getestet werden können. Hilfreich ist dabei der Ansatz des Test-Driven Developments.

Generell gilt für Clemson jedoch, dass je kleiner ein Service ist, umso so größer ist der Anteil von Koordinationslogik und Kommunikationslogik zu externen Abhängigkeiten in Relation zur internen Domain-Logik. Besteht der Service zum größten Teil aus Koordinations- und Kommunikationslogik, kann es Sinn ergeben, auf Unit-Tests zu verzichten. In diesen Fällen soll der Service mit Component-Tests getestet werden, die später erläutert werden.

Generell sollten Unit-Tests immer hinterfragt werden und zwar dahingehend, ob der Aufwand genug Nutzen bringt. Unit-Tests können die Entwicklung auch beeinträchtigen, wenn sie zu viel Aufwand kosten.

Es gibt für Clemson zwei unterschiedliche Arten von Unit-Tests, welche er von Jay Fields übernommen hat [Fie14]. Bei den “Sociable unit testing" liegt der Hauptaugenmerk darauf, das Verhalten des zu testenden Moduls mithilfe der Beobachtung von Zustandswechseln zu testen. Dabei handelt es sich um ein Blackbox-Testing über die Schnittstelle des Moduls. Die andere Art ist das “Solitary unit testing". Hierbei werden die Abhängigkeiten simuliert, damit die Interaktionen und Zusammenarbeit zwischen einem Objekt und seine Abhängigkeiten untersucht werden können.

Die verschiedenen Unit-Test Arten sind für unterschiedliche Module innerhalb des Services gedacht. So soll das Domain Module, welches die Logik und die Berechnungen durchführt und verschiedene Zustände einnehmen kann, mit Sociable Unit-Tests getestet werden. Nach Clemson gibt es kaum Vorteile dieses Modul zu isolieren, da es hauptsächlich zustandsorientiert ist. Daher sollten keine Abhängigkeiten oder Ähnliches simuliert werden.

Bei Modulen, die die Kommunikation mit externen sowie internen Abhängigkeiten übernehmen, sollen Solitary Unit-Tests verwendet werden. In seinem Schema sind das die Module Resources, Service Layer, Repositories und Gateways. Dort ist es am sinnvollsten die Abhängigkeiten zu simulieren, was die Tests auch effizienter macht. Mithilfe der Simulation der Abhängigkeiten können diese Module zuverlässig und wiederholbar getestet werden. Diese Module mit reellen Abhängigkeiten zu testen ist aufwendiger.

Integration Testing

Bei den Integration-Tests sollen die Interaktionen und Kommunikationspfade zwischen den beteiligenden Komponenten getestet werden. Dafür wird ein Ausschnitt vom Gesamtsystem aus mehreren Komponenten zusammen getestet. Dabei kann der Testumfang unterschiedlich groß sein. Unit-Tests haben davor nur die einzelnen Komponenten isoliert getestet, aber nicht die Zusammenarbeit untereinander.

Bei einem Microservice bedeutet dies, dass die Kommunikation zwischen den Modulen, welche mit externen Komponenten interagieren, und den externen Komponenten getestet

werden. Das Ziel ist es zu prüfen, dass die Module mit externen Komponenten erfolgreich kommunizieren können. Es geht dabei nicht darum, die externe Komponente zu testen.

Die Integration-Tests können bei Refactoring Maßnahmen oder beim Ausbauen der Funktionalitäten sehr hilfreich sein. Dafür bieten sie schnelles Feedback an. Jedoch können die Tests aus mehreren Gründen fehlschlagen. Es kann einen Fehler im Modul geben oder die externe Komponente ist fehlerhaft oder einfach nicht vorhanden. Daher sollten nur einige wenige Tests von diesem Typ geschrieben werden. Es kann sogar Sinn ergeben, die Integration-Tests aus der Deployment Pipeline zu entfernen, damit das Veröffentlichen nicht blockiert wird. Wird eine weitere Testabdeckung benötigt, können auch Unit-Tests oder Contract Tests verwendet werden.

Auch hier unterscheidet Clemson zwei verschiedene Arten von Integration-Tests. Der Gateway Integration-Test hat das Ziel die Module zu testen, welche mit externen Services via HTTP kommunizieren. Dabei soll das Verhalten bei verschiedenen Fehlern wie z.B. falschen Headern oder Bodys getestet werden. Dazu soll auch das Verhalten bei typischen Netzwerkfehlern, wie Timeouts, überprüft werden. Sollten bei einigen Fehler in der Kommunikation spezielles Verhalten notwendig sein, dann sollte dies auch unbedingt getestet werden. Damit das gewünschte Verhalten im Fehlerfall auch sicher auftritt. Um das Verhalten bei Fehlern in der Kommunikation und andere Schwierigkeiten wie Timeouts und lange Antwortzeiten zu testen, ist es hilfreich, die externe Komponenten durch Stubs zu ersetzen. Dadurch können diese Umstände leichter simuliert werden.

Der andere Integration-Test ist der Persistence Integration-Test. Hierbei soll die Kommunikation mit externen Datenspeicher, wie z.B. Datenbanken, getestet werden. Bei der Verwendung von Frameworks, welche objektrelationale Abbildungen (ORM) anbieten, wie z.B. Hibernate, geht es auch darum zu testen, dass die konfigurierten Mappings mit den zurückgegebenen Werten aus den Datenbanken übereinstimmen. Da viele solcher Frameworks sehr anspruchsvoll sind, was die Kommunikation angeht, ist es wichtig, dass geöffnete Transaktionen geschlossen werden, bevor der eigentliche Test startet. Dadurch wird die komplette Kommunikation getestet. Darauf aufbauend liegen externe Datenspeicher oft auf anderen Computern. Deswegen muss auch das Verhalten bei Timeouts und langsamen Antworten getestet werden.

Component Testing

Unter Component Testing bei einer Microservice Architektur versteht Clemson das isolierte Testen eines Services. Der Service wird dahin gehend isoliert, dass zu einem die externen Abhängigkeiten isoliert werden und zum anderen interne Schnittstellen verwendet werden, die für die Konfiguration des Services gedacht sind. Die Tests haben das Ziel, das gekapselte

Verhalten des Services zu testen. Dafür sollen die Tests die externe Schnittstelle aus der Sicht des Services anfragen, der den Service verwenden möchte und somit das Verhalten testen. Die Isolierung bringt die Vorteile, den Service kontrolliert testen zu können, sowie komplexes Verhalten in Verbindungen mit externen Komponenten zu vermeiden.

Wie bei den Testarten davor auch, gibt es für Clemson zwei verschiedene Wege die Component Tests aufzubauen. Beim ersten Weg wird der Service sozusagen von innen heraus getestet. Er nennt dies In-Process Component Test. Hierbei sollen innerhalb des Services Test Doubles verwendet werden, welche die Aufgaben haben, externe Komponenten zu simulieren. Hiermit soll die Verwendung des Netzwerkes vermieden werden, da dies einen Geschwindigkeitsvorteil verspricht. Dieser Weg setzt jedoch voraus, dass der Service in einem Test-Modus gestartet werden kann, bei dem die externen Komponenten intern simuliert werden.

Die andere Möglichkeit ist, die Abhängigkeiten außerhalb des Services zu simulieren. Im Detail heißt das, dass die externen Komponenten mithilfe eines Stubs simuliert werden. Dadurch muss der Service nicht in einem Test-Modus oder ähnlichem gestartet werden, sondern nur gestartet werden. Der Service erhält bei diesem Ansatz über das reelle Netzwerk Anfragen, welcher er bearbeiten muss. Dies kann die Dauer der Tests erhöhen. Sollte der Service jedoch eine komplexe Logik zum Starten oder ähnliches besitzen, dann kann diese Möglichkeit die bessere sein.

Contract Testing

Nach den bisherigen Tests kann angenommen werden, dass der Service die geforderten Funktionalitäten erfüllt. Dabei gibt es von Clemson keine Einteilung zwischen funktionalen und nicht-funktionalen Anforderungen.

Für jede Kommunikation zwischen zwei Services existiert ein Kontrakt. Diese Kontrakte werden mit Contract Tests getestet. Wichtig ist dabei anzumerken, dass sie nicht das Verhalten der einzelnen Services testen.

Dafür empfiehlt Clemson Consumer Driven Contract Test. Dieses Testverfahren wurde bereits im Grundlagenkapitel (Abschnitt 3.1.4) dieser Arbeit vorgestellt.

Da der Kontrakt aus der Sicht des Consumers definiert wird, sollen diese Tests auch von dem Team entworfen werden, welche den Consumer entwickelt. Dabei sollen diese Tests unabhängig von anderen Contract Tests sein. Mit einer Einbindung in die Deployment Pipeline können der Consumer sowie der Provider überprüfen, ob ihre Änderungen zu Verletzungen des Kontraktes führen.

Der genaue Ablauf dieser Tests ist abhängig von dem verwendeten Test-Werkzeug. Dafür schlägt Clemson PACT, PACTO und Janus vor.

End-To-End Testing

Beim End-To-End Testing geht es darum, die gesamte Anwendung von einem Ende bis zum anderen Ende zu testen. Ziel ist hierbei zu testen, ob die Anwendung in ihrer Gesamtheit alle Anforderungen einhält.

Die Tests können über eine grafische Schnittstelle erfolgen oder über jeweilige Serviceschnittstellen. Besonders bei einer Anwendung mit der Architektur Microservices soll getestet werden, ob die Zusammenarbeit der einzelnen Services funktioniert. Es gibt zusätzliches Vertrauen, dass die Anwendung funktioniert und die Konfigurationen wie FireWalls und Proxies korrekt ist.

Diese Tests sind Blackbox-Tests. Mit dem Überwachen von Zustandswechseln und Auslösung von Events wird die Korrektheit der Anwendung überwacht.

Normalerweise sollten alle Services sich im Testbereich der End-To-End Tests befinden. Es kann jedoch Szenarien geben, in denen es sinnvoll ist, einige externe Services zu simulieren. Dies trifft besonders zu, wenn die Services von einem Drittentwickler stammen. In diesen Fällen können diese Services nur schwer für diese Tests konfiguriert werden. Durch das Simulieren von externen Services könnte zwar das Vertrauen in die Gesamtanwendung sinken, aber die Tests können stabiler ablaufen.

Das Entwerfen von End-To-End Tests kann sehr schwierig sein. Aufgrund der Vielzahl von Modulen und Services, die miteinander agieren, können leicht instabile Tests entstehen. Instabil bedeutet hier, dass die Tests mal erfolgreich sind und mal nicht ohne, dass eine Änderung vorgenommen wurde. Besonders asynchrone Aufrufe können diese Tests erschweren. Daher gibt Clemson fünf Empfehlungen aus, End-To-End Tests zu schreiben.

Die erste Empfehlung rät dazu, nur wenige End-To-End Tests zu schreiben. Anforderungen an die Software können besser auf den unteren Testebenen abgedeckt werden. Bei End-To-End Tests geht es darum zu prüfen, ob die generelle Zusammenarbeit funktioniert. Eine Möglichkeit um End-To-End Tests zu begrenzen, ist es eine Höchstdauer zu definieren, wie lange die End-To-End Tests insgesamt laufen dürfen. Diese Einschränkung hilft auch dabei End-To-End Tests auszusortieren.

Als nächstes empfiehlt Clemson, sich auf personas und user journeys zu konzentrieren. Dadurch kann das Vertrauen in die Anwendung erhöht werden, dass die Hauptfunktionen ihre Anforderungen erfüllen.

Dazu rät Clemson die Ende der Tests gut zu wählen. Verhält sich eine Komponente wie ein externer Service oder die GUI sehr instabil bei den Tests, so dass die Tests immer wieder scheitern, obwohl kein Fehler vorliegt, dann sollten diese Komponenten nicht Teil dieser Tests sein.

Des Weiteren schlägt Clemson die Nutzung von Infrastructure-as-Code vor. Dabei geht es darum, die benötigte Infrastruktur via Source-Code bereit zu stellen und nicht mit manuellen Verfahren. Dadurch können neue Umgebungen schneller aufgebaut werden und dies vor allem in einer reproduzierbaren Weise.

Als letztes empfiehlt Clemson unabhängige Testdaten. Bei der Verwendung von bereits existierenden Daten können viele Fehler auftreten, wie z.B. dass ein Datensatz nicht mehr vorhanden ist oder nicht dem erwarteten Wert entspricht. Können die Services ihre eigenen Daten verwalten und unterstützen die Erstellung der Daten über ihre Schnittstellen, können die End-To-End Tests die Testdaten vor der Ausführung der Tests selber definieren. Andernfalls müssen die Daten vorher in die Datenbank importiert werden.

Exploratory Testing

Mit Exploratory Testing versteht Clemson das manuelle Testen der Anwendung. Es ermöglicht die Anwendung kennen zu lernen. Damit können unter anderem End-To-End Tests weiterentwickelt werden. Dafür werden die gewonnenen Erkenntnisse aus diesen Tests für das Entwerfen von End-To-End Tests verwendet.

4.2.4 Testkonzept von Google

Im folgenden Abschnitt wird das Testkonzept von Google vorgestellt, welches im Buch “How Google Tests Software" präsentiert wird. Der Autor des Buches ist James Whittaker [WAC12].

Google teilt seine Tests in drei verschiedene Testarten ein. Es gibt Small, Medium und Large Tests. Die Namen von den Tests sind nicht von großer Bedeutung, sondern können auch andere Namen tragen. Wichtig ist nur, dass die Definitionen und Ziele der Tests weiterhin gelten.

Google stellt seine Testarten innerhalb eines Trapezes vor. Dabei gelten ähnliche Aussagen wie bei den vorherigen Testpyramiden. Je weiter unten sich eine Testart befindet, umso mehr Tests sollen von dieser Art vorhanden sein. Zudem ist die Isolierung und die Ausführungsdauer bei Small Tests am geringsten. Mit jeder Stufe nach oben wächst das Vertrauen in die Gesamtanwendung.

Google entscheidet für jedes Produkt spezifisch, was getestet werden soll und in welcher Intensität. Dieser Prozess ist zudem ein sehr dynamischer Prozess, so dass es Änderungen geben kann.

Über alle Testarten hinweg werden automatisierte und manuelle Tests vermischt. Das bedeutet, dass es bei allen Testarten auch manuelle Tests geben kann. Der genaue Mix hängt dabei von den genauen Testschwerpunkten ab.

Grundsätzlich gilt bei Google jedoch, dass alles, was automatisiert getestet werden kann und keine menschliche Intuition oder ähnliches benötigt, auch automatisiert getestet werden soll. Wird jedoch menschliche Intuition benötigt, dann kann der Test in Form eines manuellen Tests vorliegen. Dies kann bei Benutzeroberflächen der Fall sein, aber auch bei Schnittstellen oder Methoden, bei denen private Daten verwendet werden.

Bei den manuellen Tests werden sowohl exploratives Testen als auch geskriptetes Testen eingesetzt. Das manuelle Testen wird mit Werkzeugen zur Automatisierung unterstützt. Das heißt, dass das manuelle Testen aufgezeichnet wird. Dabei werden die Interaktionen und deren Validierungen aufgezeichnet. Auf Grundlage dieser Aufzeichnungen werden aus den manuellen Tests automatisierte Tests erstellt. Google verfolgt dabei den Gedanken, dass sich die Tester bei manuellen Tests immer auf neue Fehler konzentrieren sollen.

Zusätzlich werden bei Google automatisierte Bug-Reports erstellt. Sollte ein automatisierter Test fehlschlagen, dann bekommt automatisch der Autor der letzten Änderung am Sourcecode die E-Mail mit dem Bug-Report. Es wird davon ausgegangen, dass am wahrscheinlichsten die letzte Änderung zum Fehler geführt hat.

In den nächsten Abschnitten werden die einzelnen Testarten vorgestellt.

Small Tests

Whittaker beschreibt jede einzelne Testart mit einer Frage. Die dazugehörige Frage für die Small Tests lautet: “Does this code do what it is supposed to do?" [WAC12]. Wie aus dieser Frage abgeleitet werden kann, sollen bei Small Tests einzelne Funktionen oder Module getestet werden. Der Fokus liegt dabei auf dem Prüfen von Funktionen und dem korrekten Verhalten bei Fehlern. Außerhalb von Google werden diese Tests auch häufig Unit-Tests genannt.

Die Small Tests sollen in einer simulierten Umgebung stattfinden. Das heißt, dass externe Abhängigkeiten durch Stubs oder Mocks ersetzt werden. Dabei haben Small Tests den kleinsten Testbereich von allen drei Testarten.

Durch die Ausführung in einer simulierten Umgebung haben die Small Tests sehr geringe Laufzeiten. Sie sollen unter einer Sekunde ablaufen und spätestens nach einer Minute abgebrochen werden. Wegen der geringen Laufzeit können Fehler schneller gefunden werden.

Die Small Tests laufen meistens automatisiert ab. In Sonderfällen kann es Sinn ergeben, dass diese Tests manuell ausgeführt werden.

Verantwortlich für das Design und Implementieren der Tests sind die jeweiligen Entwickler der Funktion, die getestet werden soll.

Medium Tests

Die Frage mit der Whittaker Medium Tests beschreibt lautet:“Does a set of near neighbor functions interoprate with each other the way they are supposed do to?" [WAC12]. Im einzelnen heißt das, dass bei Medium Tests das Interagieren zweier Funktionen bzw. Module überprüft werden soll. Der Fokus liegt dabei auf dem Testen zweier interagierenden Funktionen. Solche direkt miteinander interagierenden Funktionen werden bei Google “nearest neighbor functions" genannt. Medium Tests werden häufig auch Integrationstests genannt.

Da nicht eine einzelne Funktion getestet werden soll, sondern das Interagieren zweier Funktionen, ist der Testbereich hier größer als bei den Small Tests. Zudem müssen nicht alle externen Abhängigkeiten simuliert werden. In den Tests können z.B. auch externe Datenspeicher, wie Datenbanken, verwendet werden.

Aufgrund der geringen Isolierung können Medium Tests eine höhere Dauer haben. Medium Tests sollen innerhalb einer Sekunde ablaufen. Spätestens nach 5 Minuten sollen sie jedoch abgebrochen werden. Wegen der längeren Laufzeit müssen Medium Tests auch nicht so häufig ausgeführt werden wie Small Tests.

Die Verantwortung für die Tests liegen bei den Entwicklern, die für die Module zuständig sind. Dies beinhaltet sowohl Entwerfen, Implementieren und Wartung der Tests.

Large Tests

Large Tests beschreibt Whittaker mit der Frage “Does the product operate the way a user would expect and produce the desired results?" [WAC12]. Die Frage verdeutlicht den Fokus, welcher bei den Large Tests gilt. Es geht darum zu prüfen, ob das Produkt oder die Features den Erwartungshaltungen des Nutzers entsprechen. Zwar werden auch die Interaktionen der Features getestet, jedoch liegt der Fokus nicht darauf. Daher sollen bei den Large-Tests drei oder mehr Features getestet werden. Dabei sollen die Tests typische Nutzer Szenarien beschreiben. Außerhalb von Google tragen Large Tests häufig die Namen End-To-End Tests oder System Tests.

Bei den Large Tests sollen keine Mocks oder Stubs verwendet werden. Das heißt, dass keine externen Abhängigkeiten simuliert werden sollen. Deswegen werden bei diesen Tests auch reelle Datenbanken sowie das Netzwerk verwendet.

Durch die fehlende Nutzung von simulierten Abhängigkeiten verlängert sich die Testdauer. Large Tests sollen so schnell wie möglich ablaufen. Nach 15 Minuten sollen sie abgebrochen werden. Hier wird auch eine Sonderform der Large Tests eingeführt und zwar Enormous Tests. Diese unterscheiden sich in der Erwartung zur Dauer von den normalen Large Tests. Die

Enormous Tests sollen zwar auch so schnell wie möglich ablaufen, aber sie sollen erst nach einer Stunde abgebrochen werden.

Dabei können die Tests sowohl in automatisierter Form als auch in manueller Form vorliegen. Wichtig dabei ist nur, dass alle Subsysteme der Anwendung von der Benutzeroberfläche bis hin zum Backend Datenspeicher getestet werden. Aus diesem Grund sind Ende-Zu-Ende Szenarien, die einen gesamten Service oder ein gesamtes Produkt testen, auch Large Tests.

4.2.5 Gegenüberstellung

In diesem Abschnitt werden die vier vorgestellten Testkonzepte miteinander verglichen. Es werden Gemeinsamkeiten sowie Unterschiede herausgearbeitet.

Dafür werden zunächst die von den Autoren gegebenen Allgemeinen Hinweise zu den Testkonzepten miteinander verglichen und im Anschluss die verschiedenen Testarten miteinander. Dafür sind in der Tabelle 4.1 die wichtigsten Merkmale zusammengefasst.

Tabelle 4.1: Gegenüberstellung Allgemein

Name	Pyramide	TDD	Unterteilung Testkonzept	Eigene Deployment Pipeline	Automatisierte Tests
Wolff	Ja	Ja	Ja	Ja	Ja
Newman	Ja	Keine Aussage	Nein	Ja	Keine Aussage
Clemson	Ja	Ja	Nein	Keine Aussage	Keine Aussage
Google	Nein	Keine Aussage	Keine Aussage	Keine Aussage	Ja

Auffallend beim Vergleichen der vier Testkonzepte ist, dass sie alle ihre Testkonzepte in sehr ähnlichen geometrischen Formen darstellen. Drei von ihnen in Form einer Pyramide und nur das Testkonzept von Google wird in Form eines Trapezes dargestellt. Dies ist dahingehend wenig überraschend, da sich Newman und Clemson auf die Testpyramide von Cohn beziehen.

Dennoch geben die geometrischen Formen ähnliche Aussagen wieder. Die Position innerhalb der Form gibt Auskunft darüber, wie viele Tests in Relation zu den anderen Testarten ausgeführt werden sollen.

Bei dem Testkonzept von Google und Newman gibt die Form auch den Grad der Isolierung und das steigende Vertrauen in die Anwendung in Vergleich zu den anderen Testarten wieder. Bei Wolff und Clemson machen die Testpyramiden keine expliziten Aussagen darüber, aber es kann aus den Beschreibungen der Testfälle ähnliche Aussagen gezogen werden.

Ein großer Unterschied von Wolff gegenüber den anderen Testkonzepten ist, dass Wolff explizit sein Testkonzept unterteilt. Es gibt ein Konzept für das Testen eines Microservices und ein Konzept für das Gesamtsystem. Genau genommen definiert Newman ähnliches, indem er eine gemeinsame Ebene für die End-To-End Tests definiert, stellt dies jedoch nicht gesondert heraus. Die anderen Testkonzepte machen das nicht. Allerdings muss angemerkt werden, dass das Testkonzept von Google kein speziell entworfenes Testkonzept für Microservices ist.

Auch beim Prozess zum Erstellen der Microservices gibt es Unterschiede. Während Wolff und Newman empfehlen, dass jeder Microservice eine eigene Deployment Pipeline besitzt, äußern sich Clemson und Whittaker nicht dazu. Da jedoch Newman das Testkonzept von Clemson evaluiert hat, liegt die Vermutung nahe, dass auch Clemson die Verwendung von eigenen Deployment Pipelines pro Microservice vorzieht.

Wolff und Newman sind sich bei ihren Deployment Pipelines einig, dass es am Ende der gemeinsame Schritt End-To-End Tests heißen soll. Sie unterscheiden sich aber bei dem Schritt davor. Wolff sieht dafür den Schritt eines gemeinsamen Integrationstests vor und bei Newman kommen die Microservices aus ihren eigenen Deployment Pipelines in den Schritt der gemeinsamen End-To-End Tests. Da Newman auch gar keine Integrationstests vorsieht, ist dieser Unterschied nicht überraschend.

Einig sind sich alle vier Konzepte dabei, dass die einzelnen Tests größtenteils automatisiert ablaufen sollen. Dabei sind sie sich zudem auch einig, dass die automatisierten Tests durch manuelle Tests ergänzt werden sollen. Clemson und Wolff sind darüber hinaus derselben Meinung, dass die Strategie Test-Driven-Development (TDD) zum Einsatz kommen sollte, um die Qualität der Tests und des Source Codes im Allgemeinen zu erhöhen.

Ergänzend dazu sagen Clemson und Whittaker, dass für jeden Microservice bzw. Produkt entschieden werden muss, wie genau und in welcher Intensität es getestet werden soll.

Auf jeden Fall definieren alle vier Konzepte verschiedene Testebenen, die durchgeführt werden sollen. Dabei gibt es einige Unterschiede und Gemeinsamkeiten in den verschiedenen Testebenen und deren Definitionen. In der Tabelle 4.2 sind die verschiedenen Testebenen bzw. Teststufen aufgeführt.

Unit-Tests

In allen vier Testkonzepten werden Unit-Tests definiert. Die Small Tests tragen intern bei Google einen anderen Namen, aber wie schon dargestellt, würden die Small Tests außerhalb von Google eher Unit-Tests heißen.

Tabelle 4.2: Gegenüberstellung Testebenen

Name	Unit-Test	Integrations-test	Service-Test	CDCT	End-To-End Tests	Manuelle Tests
Wolff	Ja	Ja	Nein	Ja	Ja	Ja
Newman	Ja	Nein	Ja	Ja	Ja	Ja
Clemson	Ja	Ja	Ja	Ja	Ja	Ja
Google	Small Test	Medium Test			Large Test	Ja

Tabelle 4.3: Gegenüberstellung der Unit-Test-Definitionen

Name	Definitionen
Wolff	Testen einzelner Bestandteile; kleinste Einheit; jede Änderung am Sourcecode abdecken
Newman	Testen einer Funktion oder Methode
Clemson	Testen der kleinsten Einheit
Google	Testen einzelner Funktionen bzw. Methoden

Die Definitionen ähneln sich stark, wie der Tabelle 4.3 entnommen werden kann. Bei allen vier sollen die kleinste Einheit bzw. kleinsten Module getestet werden. Dabei nennen alle explizit oder als Beispiel Funktionen und Methoden, die getestet werden sollen.

Bei genauerer Betrachtung der Definitionen hebt sich Clemson von den anderen Definitionen dahingehend ab, dass er die Unit-Tests in zwei verschiedenen Arten unterscheidet. Clemson geht dabei sehr detailliert auf die einzelnen Komponenten eines Microservices ein. Dies machen die anderen drei Testkonzepte nicht.

Die Unterscheidung von verschiedenen Unit-Tests hat Auswirkungen darauf, wie einzelne Module bzw. Funktionen getestet werden sollen. Während Whittaker, Newman und Wolff empfehlen, die Unit-Tests komplett isoliert ablaufen zu lassen, rät Clemson dazu, Unit-Tests, welche zu “Sociable Unit Tests" gehören, nicht zu isolieren. Dazu gehören Module wie die Domain Logik, bei der nach Clemson das Nutzen der Isolierung zu gering wäre. Aufgrund der im Vergleich zu Clemson nicht so detaillierten Sichtweise der anderen Testkonzepte auf einen Microservice, ist es schwierig ein Vergleich zu ziehen.

ClemsoService-Testn schleißt sich der Idee der vollständigen Isolierung der Unit-Tests bei den “Solitary Unit Tests" an. Von der Isolierung versprechen sich alle einen Vorteil in der Ausführungsdauer der Tests und deren Stabilität.

Wolff beschreibt die Isolierung der Unit-Tests als eine Voraussetzung für die Unit-Tests. Damit hebt sich Wolff nicht nur von Clemson, sondern auch von Newman und Whittaker ab, da keiner der drei die Isolierung der Tests als Voraussetzung benennt.

Bei der Dauer der Unit-Tests herrscht größtenteils Einigkeit. Für Wolff, Newman und Whittaker sind die Unit-Tests die Tests mit der geringsten Dauer und sollen sehr schnell sein. Dabei definiert nur das Testkonzept von Google, was schnell bedeutet. Google schreibt explizite Soll-Dauer und Höchst-Dauer vor. Clemson äußert sich zur Dauer überhaupt nicht.

Dazu passt auch, dass alle bis auf Whittaker festlegen, dass die Unit-Tests automatisiert sein sollen. Nur beim Testkonzept bei Google wird gesagt, dass es Sonderfälle geben kann, wo das manuelle Ausführen von Unit-Tests Sinn ergeben kann.

Wolff und Whittaker legen bei ihren Konzepten auch fest, wer für die Unit-Tests zuständig ist. Bei beiden sind es jeweils die Entwickler der zu testenden Funktion. Newman und Clemson haben über Zuständigkeiten keine Aussage getroffen.

In dem Konzept von Newman werden die Unit-Tests als Grundlage für jede Änderungen am Sourcecode beschrieben. Auch Wolff sieht vor, dass jede Änderung am Sourcecode mit einem Unit-Test abgedeckt werden soll. Schon im Gegensatz dazu steht die Aussage von Clemson, dass Unit-Tests nur sinnvoll sind, wenn genug Domain Logik vorliegt. Dies bedeutet für ihn, dass der Service nicht hauptsächlich aus “Plumbing Code" besteht. Mit “Plumbing Code" ist Sourcecode gemeint, der Ressourcen von außen an die Domain Logik weiterleitet und wieder nach außen abgibt. Ein Beispiel ist das Modul, dass die Datenbankanbindung verwaltet.

In der Tabelle 4.4 sind die einzelnen Punkte nochmal zusammengefasst.

Tabelle 4.4: Gegenüberstellung Unit-Test

Name	Isoliert	Schnellste Testart	Zeitliche Grenzen	Entwickler Zuständig	Ein-teilung	Manuelle Ausführung
Wolff	Ja	Ja	Nein	Ja	Nein	Nein
Newman	Ja	Ja	Keine Aussage	Nein	Nein	Nein
Clemson	Ja und Nein	Nein	Keine Aussage	Keine Aussage	Ja	Nein
Google	Ja	Ja	Ja	Ja	Nein	Ja

Integrationstest

Bei den Integrationstests gibt es größere Unterschiede als bei den Unit-Tests. So definiert Newman in seinem Konzept überhaupt keine Integrationstests. während Wolff zwei verschiedene Arten von Integrationstests definiert.

Tabelle 4.5: Gegenüberstellung der Integrationstest-Definitionen

Name	Definitionen
Wolff	Testen der Zusammenarbeit mehrerer Microservices
Newman	
Clemson	Testen der Interaktionen und Kommunikationspfade zwischen beteiligten Komponenten; Testen externer Module; Kein Test der Zusammenarbeit von mehreren Microservices
Google	Testen zwei interagierender Funktionen

In der Tabelle 4.5 kann leicht erkannt werden, dass sich die Definitionen für die Integrationstests teilweise sehr stark unterscheiden. Bei Google heißen die Integrationstests intern Medium Tests. Sie haben das Ziel das Zusammenwirken von interagierenden Funktionen bzw. Modulen zu testen. Clemson definiert Integrationstests ähnlich und zwar sollen bei den Integrationstests die Kommunikationspfade und Interaktionen zwischen beteiligten Komponenten überprüft werden. Dabei grenzt Clemson dies insofern ein, dass nur die Zusammenarbeit mit externen Modulen eines Microservices getestet werden soll. Es soll explizit nicht die Zusammenarbeit mit anderen Microservices getestet werden. Dies ist bei Wolff gegenteilig definiert. Bei Wolff sollen die Integrationstests die Zusammenarbeit der verschiedenen Microservices testen. Für ihn soll dies auf zwei verschiedenen Ebenen passieren. Einmal nur für einen Microservice, bei dem die anderen Microservices simuliert werden sollen, und einmal für das Gesamtsystem, wobei nur die Erreichbarkeit untereinander getestet werden soll.

Auch bei Clemson gibt es eine Unterscheidung der Integrationstests, jedoch nicht auf der Ebene, ob ein Microservice oder Gesamtsystem getestet wird, sondern welche Komponente getestet werden soll. Er unterscheidet einmal in die Module, welche die Kommunikation mit anderen Services übernehmen, und in die Module, welche die Kommunikation mit externen Datenspeichern übernehmen. Bei dem Testkonzept von Google gibt es keine Unterscheidung innerhalb der Medium Tests.

Whittaker, Clemson und Wolff sind sich einig, dass bei den Integrationstests andere Module bzw. Services, die nicht getestet werden sollen, simuliert werden sollen. Im Testkonzept von Google heißt es, dass die Medium Tests isoliert sein sollen. Wolff empfiehlt für die Simulation

explizit die Verwendung von Stubs. Auch Clemson rät zur Verwendung von Stubs. Bei Whittaker gibt es keine Empfehlungen.

Clemson sowie Wolff raten zur Verwendung von Consumer Driven Contract Tests, um die Integrationstests zu ergänzen. Auch Newman rät zur Nutzung von CDCT. Dies jedoch nicht als Ergänzung zu Integrationstest, da er keine Integrationstests vorsieht. In seinem Konzept werden die Integrationstests durch Service-Tests und vor allem durch CDCT ersetzt.

Für Clemson kann es Sinn ergeben, die Integrationstests aus der Deployment Pipeline zu entfernen. Besonders wenn die Tests zu instabil sind. Die Integrationstests können dann im Anschluss der Pipeline stattfinden. Innerhalb der Deployment Pipeline sollen die Integrationstests durch Unit-Tests und CDCT ersetzt werden. Bei Wolff sind die Integrationstests fester Bestandteil der Deployment Pipeline. Auch bei Google ist keine Ersetzung der Medium Tests vorgesehen.

Nur in dem Testkonzept von Google wird eine feste Dauer für die Tests definiert. Bei den anderen Testkonzepten gibt es nur oberflächliche bzw. keine Aussagen dazu.

Ähnlich wie bei der Testdauer, definiert auch nur das Testkonzept von Google, wer für die Tests zuständig ist. Für Google sind die Entwickler der zu testenden Funktionen und Module für das Entwerfen und Warten der Tests zuständig.

Auch in diesem Abschnitt sind die wichtigsten Merkmale zusammengefasst. Die Zusammenfassung ist in der Tabelle 4.6 dargestellt.

Tabelle 4.6: Gegenüberstellung Integrationstest

Name	Einteilung	Isoliert	CDCT	Zeitliche Grenzen	Entwickler Zuständig
Wolff	Nein	Ja	Ja	Nein	Nein
Newman					
Clemson	Ja	Ja	Ja	Nein	Nein
Google	Nein	Ja	Nein	Ja	Ja

Service-Tests

Die Testart Service-Tests taucht nur in den Konzepten von Clemson und Newman auf. Beide definieren Service-Tests sehr ähnlich, siehe Tabelle 4.7. Bei beiden geht es darum, einen Service isoliert zu testen, ob er die gewünschten Anforderungen erfüllt. Bei Clemson heißt dieser Test jedoch Component Test.

Tabelle 4.7: Gegenüberstellung der Service-Tests-Definitionen

Name	Definitionen
Wolff	
Newman	Testen eines einzelnen Services und seiner Funktionen
Clemson	Isoliertes Testen eines Services
Google	

Wie schon in der Definition erwähnt, sollen die Service-Tests bei beiden isoliert stattfinden. Dahingehend sollen andere Services simuliert werden. Jedoch nicht externe Komponenten eines Services, wie z.B. Datenbanken.

Allerdings differenziert Clemson im Gegensatz zu Newman den Service-Test genauer. Er beschreibt zwei unterschiedliche Wege den Service zu testen. Dabei unterscheiden sich beide Wege darin, wie der Service isoliert werden kann. So eine Unterscheidung findet bei Newman nicht statt.

Die einzelnen Aspekte sind in der Tabelle 4.8 zusammengefasst.

Tabelle 4.8: Gegenüberstellung Service-Tests

Name	Einteilung	Isoliert
Wolff		
Newman	Nein	Ja
Clemson	Ja	Ja
Google		

End-To-End Tests

End-To-End Tests werden wieder von allen vier Testkonzepten definiert, wenn auch mit unterschiedlichen Namen. Wolff nennt End-To-End Tests UI Tests und bei Google werden sie Large Tests bzw. Enormous Tests genannt.

Bei den Definitionen gibt es nur im Detail Unterschiede, wie der Tabelle 4.9 entnommen werden kann. Wolff definiert seine End-To-End Tests bzw. UI Tests so, dass bei diesen Tests die Oberfläche getestet wird. Das Ziel ist zu prüfen, ob die Oberfläche wie gewünscht funktioniert. Ähnlich ist auch die Definition von Newman. Bei ihm ist das Ziel der Tests, die Anwendung über die Oberfläche zu testen und zu prüfen, ob die Anwendung die Anforderungen erfüllt. Bei Clemson soll die Anwendung von Anfang bis Ende getestet werden. Dabei können die Enden selbst definiert werden. Zudem ist das Testen nicht auf die Oberfläche begrenzt. Bei Clemson

Tabelle 4.9: Gegenüberstellung der End-To-End Tests-Definitionen

Name	Definitionen
Wolff	Testen, ob die Oberfläche funktioniert
Newman	Testen der Anwendung über die Oberfläche oder Schnittstelle
Clemson	Testen von Anfang bis Ende; Testen ob alle Anforderungen erfüllt werden
Google	Verhält sich die Anwendung, wie erwartet

können End-To-End Tests auch ohne die Oberfläche stattfinden. Bei Google heißt es bei den Large Tests nur, dass getestet werden soll, ob die Anwendung die Anforderungen erfüllt.

Einig sind sich Clemson, Wolff und Newman dabei, dass diese Tests sehr fragil und instabil sein können. Dahingehend sagen Newman und Clemson, dass es nur sehr wenige End-To-End Tests geben soll. Newman nennt als maximale Anzahl eine niedrige zweistellige Zahl. Newman rät dazu gänzlich auf End-To-End Tests zu verzichten und diese durch CDCT und Unit-Tests zu ersetzen. Nichtsdestotrotz sollen End-To-End Tests auch bei Newman verwendet werden, solange es keinen adäquaten Ersatz gibt. Im Testkonzept von Google werden keine Aussagen zu der Anzahl und der Stabilität der Tests getroffen.

Die jeweiligen End-To-End Tests sollen sich auf bestimmte Bereiche konzentrieren. Bei Newman sollen die Tests Kernfunktionen testen. Während bei Clemson sich die Tests auf personas und user journeys fokussieren sollen. Ähnlich sieht das auch beim Testkonzept von Google aus. Hier sollen sich die Tests auf User Stories konzentrieren. In dem Konzept von Wolff gibt es keine Aussage dazu.

Nur bei Newman und Whittaker wird etwas zur Dauer der Tests geschrieben. Newman sagt einfach, dass diese Tests mehrere Stunden dauern können. Darauf begründet er neben der bereits erwähnten Instabilität auch seine Ablehnung gegenüber End-To-End Tests. Google hat eine offenere Definition der Dauer. Die Tests sollen so schnell wie möglich ablaufen. Es existieren jedoch Obergrenzen dafür.

Nur in dem Konzept von Clemson wird erwähnt, dass es Sinn ergeben kann, einzelne Komponenten zu simulieren. Besonders bei externen Komponenten von Dritten kann es zweckmäßig sein, diese zu simulieren.

Die verschiedenen Aspekte sind in der Tabelle 4.10 komprimiert.

Tabelle 4.10: Gegenüberstellung End-To-End Tests

Name	Isoliert	Zeitliche Grenzen	Anzahl	Fokus	Vermeidung	Instabil
Wolff	Nein	Nein	Sehr wenige	Keine Aussage	Nein	Ja
Newman	Nein	Nein	sehr niedrige zweistellige Zahl	Kern-funktionen	Ja	Ja
Clemson	Nein	Nein	Sehr wenige	personas und user journeys	Nein	Ja
Google	Nein	Ja	Keine Aussage	user stories	Nein	Keine Aussage

Manuelle Tests

Bis auf das Testkonzept von Newman sehen alle Testkonzepte manuelle Tests vor. Die Aufgabe der manuellen Tests wird jedoch in jedem Konzept anders definiert.

Bei Wolff sollen die manuellen Test dazu dienen neue Features zu testen. Im Konzept von Clemson sollen die manuelle Tests den Zweck erfüllen, die Anwendung kennen zu lernen und auf Grundlage dieses Wissens End-To-End Tests zu entwerfen. Ähnlich sieht es beim Konzept von Google aus. Hier sollen die manuellen Tests aufgezeichnet werden, um daraus automatisierte Tests zu erstellen. Jeder manuelle Test soll bei Google nur einmal durchgeführt werden. Manuelle Tests werden bei Google vor allem dort verwendet, wo menschliches Denken notwendig ist.

Wann diese Tests durchgeführt werden sollen, definiert nur Wolff. Er sagt, dass die Tests nach dem Abschluss der Deployment Pipeline ausgeführt werden sollen, damit der Deployment-Prozess nicht verzögert wird.

In der Tabelle 4.11 sind die vergleichenden Merkmale nochmal zusammengefasst.

Sonstiges

Zusätzlich zu den bisherigen Aussagen zum Testen, geben manche Konzepte noch weitergehende Hinweise mit.

Im Konzept von Wolff spielt das Loggen und das Monitoring eine sehr große Rolle. Für ihn kann nur erfolgreich getestet werden, wenn es ein ausführliches Logging und Monitoring gibt. Auch Newman erwähnt das Monitoring als nützliches Werkzeug beim Testen.

Tabelle 4.11: Gegenüberstellung manuelle Tests

Name	Aufgabe	Zeitpunkt
Wolff	Testen neuer Features	Nach dem Abschluss der Deployment Pipeline
Newman		
Clemson	Anwendung kennenlernen; Grundlage für End-To-End Tests	Keine Aussage
Google	Grundlage für automatisierte Tests; Testen, wo menschliches Denken notwendig ist	Keine Aussage

Bei Newman ist das Deployen der Services sehr wichtig. Er stellt zwei verschiedene Ansätze vor, um neue Versionen eines Services zu veröffentlichen und diese durch eine vorherige Version zu ersetzen, wenn die neue Version Fehler enthält.

Wolff und Newman erwähnen beide Performance-Tests. Besonders für Newman spielen diese Tests eine wichtige Rolle. Daher beschreibt er diese Tests sehr ausführlich.

4.2.6 Bewertung der Testkonzepte

Nach ausführlicher Betrachtung der Testkonzepte kann nicht gesagt werden, dass eins dieser Konzepte das Beste ist. Jedes Testkonzept besitzt seine Vor- und Nachteile.

Das Testkonzept von Wolff besitzt seinen Vorteil in der Unterteilung des Testkonzeptes. Mit dieser Unterteilung reagiert er sehr gut auf die Charakteristika der Architektur Microservices, dass zum einen jeder Microservice ein eigenständiges Programm ist, aber zum anderen erst die Zusammenarbeit das Gesamtsystem ergibt. Dazu sind die genannten Punkte bei der Definition von Unit-Tests nachvollziehbar.

Unverständlich ist bei seinem Testkonzept, dass er keine internen Integrationstests definiert. Er versteht unter Integrationstests immer das Testen der Zusammenarbeit der einzelnen Microservices. Damit klammert er einen wichtigen Punkt beim Testen eines Microservices aus. Auch ein Microservice kann komplexe Logik enthalten, die erst durch das Zusammenwirken von mehreren Methoden funktioniert. Dies wird bei ihm mit getestet, aber auf einer höheren Ebene mit mehr Testumfang. Dadurch ist es schwieriger, Fehler zu finden.

Beim Testkonzept von Newman ist das Definieren von Service-Tests mit CDCT ein Vorteil. Mit den Service-Tests wird sichergestellt, dass der Service seine Anforderungen erfüllt und sich wie erwartet verhält. Dazu wird durch CDCT sichergestellt, dass der Service die Kontrakte einhält.

Newmans Testkonzept fällt besonders durch das Weglassen von Integrationstests auf. Dies ist eine sehr mutige Entscheidung, da sie auf der Annahme beruht, dass die Unit-Tests und Service-Tests für genügend Abdeckung sorgen. Daher gilt hier der gleiche Kritikpunkt wie bei Wolffs Definition von Integrationstests.

Widersprüchlich wird das Testkonzept von Newman bei den End-To-End Tests. Er sammelt sehr viele Argumente, warum aus seiner Sicht keine End-To-End Tests existieren sollen, aber Ende des Kapitels definiert er sie trotzdem. Obwohl der Hinweis gegeben wird, dass die End-To-End Tests langfristig ersetzt werden sollen, wirkt dies widersprüchlich.

Auf jeden Fall ist die Aussage, keine End-To-End Tests einzusetzen, diskussionswürdig. Trotz der vielen Nachteile bieten End-To-End Tests auch große Vorteile, wie die Sicherheit, dass getestete Anwendungen ihre Anforderungen erfüllen. Insbesondere sind End-To-End Tests für Personen, die nicht Entwickler sind, am verständlichsten, weil die End-To-End Tests testen die Anwendung wie, der Endnutzer sie verwendet.

Das Alleinstellungsmerkmal vom Testkonzept von Clemson ist, dass es sehr detailliert und kleinteilig ist. Dabei definiert er einzelne Tests erst allgemein und dann spezifisch für die Architektur Microservices. Generell macht sein Konzept einen sehr durchdachten Eindruck. Im Gegensatz zu Wolff und Newman werden aus meiner Sicht keine wichtigen Tests weggelassen.

Clemsons detaillierte Beschreibung ist jedoch auch gleichzeitig sein größter Nachteil. Besonders bei der Definition von Unit-Tests ist er zu detailliert. Er geht davon aus, dass jeder Microservice grundsätzlich mit den gleichen Komponenten aufgebaut ist. Diese Annahme kann für das MARS Framework nicht übernommen werden.

Auf Grundlage dieser Annahme basiert auch die Einteilung Clemsons der Unit-Tests. Besonders die Entscheidung bei “Sociable unit testing" die Tests nicht zu isolieren, ist zu kritisieren. Für mich stellt sich die Frage, ob es noch dann Unit-Tests sind, denn aufgrund der Nicht-Isolierung der Tests können auch andere Komponenten mit getestet werden. Daraus generiert sich die Frage. Wird noch die einzelne Einheit getestet oder das Zusammenwirken?

Weiter negativ aufgefallen ist, dass Clemson für die Einteilung der Tests keine Beispiele gegeben hat. Es wird zwar ausführlich beschrieben, welche Komponente mit welchem Tests getestet werden soll. Trotzdem wären explizite Beispiele sehr hilfreich, um einschätzen zu können, wie die Tests in der Praxis aussehen.

Das Testkonzept von Whittaker steht dazu im krassen Gegensatz, was die Details angeht. Die grobe Definition der Tests, ausschließlich nach ihrer Aufgabe, ist gleichzeitig Vor- und Nachteil.

Aufgrund der Definition von Whittaker können die Tests für jedes System verwendet werden, weil es immer einzelne Komponenten oder zusammenhängende Methoden existieren werden. Dabei kristallisiert sich auch heraus, dass das Testkonzept für alle Google Produkte gelten soll.

Die grobe Definition lässt jedoch auch sehr viel Spielraum zu. Dies kann immer dazu führen, dass die Tests anders verwendet werden, als ihr ursprüngliches Ziel ist. Dazu besteht die Gefahr, dass bei Detailfragen nicht klar ist, wie reagiert werden soll.

Ein großer positiver Aspekt im Testkonzept von Google sind die definierten Grenzen bei der Dauer der Tests. Sie unterstützen dabei, Tests klein zuhalten, da zu große Tests an der Grenze scheitern.

Negativ bei den meisten Testkonzepten ist, dass sie keine Unterscheidung zwischen funktionalen und nicht-funktionalen Anforderungen machen. Nur bei Newman existiert eine deutliche Unterscheidung. Dazu geht Wolff halbwegs darauf ein, indem er Last- und Kapazitätstests definiert, aber eine explizite Unterscheidung gibt es nicht.

4.3 Exploratory Testing

Innerhalb dieses Abschnittes soll untersucht werden, ob und inwiefern die Verwendung von Exploratory Testing bei einer Anwendung, die Microservices nutzt, sinnvoll ist. Diese Fragestellung soll in verschiedenen Szenarien untersucht werden. Die Szenarien bilden dabei die verschiedenen Arten von Änderungen ab. Mit Ausnahme der Änderungen, die sich auf der Ebene der Architektur befinden.

Wie im Abschnitt zu den MARS spezifischen Herausforderungen beschrieben, existieren für das MARS Framework keine Anforderungen. Erschwerend kommt hinzu, dass besonders die Anforderungen, die im Kontext der Simulation stehen, generisch sind. Denn diese sind abhängig von der Simulation und können sich von Simulation zu Simulation unterscheiden. Besonders unter diesen Aspekten wurde Exploratory Testing ausgesucht.

Es existieren verschiedene ähnliche Verfahren, wie z.B. das Error Guessing. Bei diesem Verfahren wird auf Grundlage von Erfahrungen und Kenntnissen versucht, Fehler zu erraten. Die vermeintlichen Fehler sollen mithilfe von Testfällen entdeckt werden [Hom13].

Die Entscheidung für Exploratory Testing begründet sich in dem größeren Umfang, den es bietet. Besonders der Aspekt, die Anwendung gleichzeitig zu testen und Informationen zu sammeln, ist ausschlaggebend für diese Entscheidung. Mit den gewonnenen Informationen soll die Definition von Anforderungen leichter fallen. Es soll hiermit die Schwierigkeit gelöst werden, dass für das MARS Framework bisher keine Anforderungen erstellt worden sind.

Bei allen Szenarien wird Exploratory Testing mit automatisierten Tests verglichen. Denn automatisierte Tests werden als Gegenstück zu Exploratory Testing verstanden. Da Exploratory Testing nicht nur aus der Ausführung von Tests besteht, sondern z.B. auch aus dem Erlernen der Anwendung und dem daraus resultierenden dynamischen Erstellen von Testfällen, werden mehrere Szenarien benötigt. In den einzelnen Szenarien sollen die verschiedenen Bereiche von Exploratory Testing beleuchtet werden.

Exploratory Testing wird ausschließlich auf der Ebene von End-To-End Tests bzw. GUI Tests untersucht. Zu einem ist es hier, am einfachsten einen Vergleich zwischen automatisierten und manuellen Tests aufzubauen, und zum anderen können die verschiedenen Arten von Änderungen leichter in Szenarien eingebaut werden.

In den Szenarien werden Tests ausgeführt, die z.B. über die Weboberfläche Dateien importieren oder bestimmte Seiten aufrufen sollen.

Im nächsten Abschnitt wird zunächst die Umgebung vorgestellt, in der die Szenarien ausgeführt werden. Im darauffolgenden Abschnitt werden die einzelnen Szenarien detailliert vorgestellt. Danach werden die Durchführungen und die Ergebnisse der Szenarien beschrieben. Darauf aufbauend findet im letzten Abschnitt eine Bewertung der Ergebnisse statt.

4.3.1 Umgebung

Im laufenden Abschnitt wird die Umgebung beschrieben, in der die Szenarien durchgeführt werden.

Als Anwendung, an der die Szenarios durchgeführt werden, wird das MARS Framework verwendet. Die für das MARS Framework entwickelte Oberfläche wurde mit dem Javascript-Framework Angular entwickelt. Dabei bildet die Weboberfläche einen eigenen Microservice, der sich in einem Docker-Container befindet.

Vom MARS Framework wird eine Version aus dem Dezember 2016 genutzt. Auf eine Aktualisierung der Anwendung wird aus verschiedenen Gründen verzichtet. Z einen aus der Erfahrung heraus, dass bei Aktualisierungen nach einem langen Zeitraum bei dem MARS Framework sehr umfangreiche Arbeiten nach sich ziehen, um die Anwendung wieder starten zu können. Zum anderen würde eine Aktualisierung der Untersuchung keinen Mehrwert bringen, da es nicht relevant ist, in welcher Version die Anwendung vorliegt. Sie dient lediglich als Beispiel.

Innerhalb der Testumgebung wird die Oberfläche nicht, wie eigentlich vorgesehen, in einem Docker-Container gestartet. Das liegt vor allem daran, dass der Container, der die Aufgabe eines Applications-Servers übernimmt, nicht stabil funktioniert. Dahingehend ist es nicht gewährleistet das MARS Framework regelmäßig zu starten.

Aus diesem Grund wurde entschiedenen die Weboberfläche direkt aus der Entwicklungsumgebung zu starten. Dabei funktionierten die eigenen Befehle von Angular zum Starten der Weboberfläche aus unbekannten Gründen nicht. Daher wird die Oberfläche mithilfe des JavaScript-Tools Gulp gestartet. Mit diesem Tool wird die Weboberfläche auch innerhalb des Docker-Containers gestartet. Nachdem erfolgreichen Starten der Anwendung ist sie im Browser unter der Adresse “localhost:8080/#/" erreichbar.

Diese Modifizierung hat keinen Einfluss auf die Ergebnisse. Bei den Szenarien wird das Zusammenwirken von mehreren Microservices nicht benötigt. Daher kann die Weboberfläche lokal gestartet werden, da eine Kommunikation mit anderen Microservices nicht unbedingt notwendig ist.

Durch die fehlende Kommunikation mit anderen Microservices kann der Import von Dateien nicht ausgeführt werden, da mit dem FileService und den dazugehörigen Import-Services nicht kommuniziert werden kann. Deswegen wurde die Importseite so angepasst, dass sie den Import als Erfolgreich bewertet, wenn alle Felder gefüllt sind und der Button für das Importieren ausgeführt worden ist. Mit dieser Änderung wird das eigentliche Verhalten simuliert. Daher hat auch diese Änderung keinen Einfluss auf die Ergebnisse, weil bei den Szenarien, die die Import-Seiten verwenden, das Verhalten der Weboberfläche im Fokus steht und nicht das Zusammenarbeiten der Microservices.

Die automatisierten End-To End-Tests werden mit dem Framework Selenium in der Programmiersprache Java entwickelt. Mit diesen automatisierten Tests wird Exploratory Testing verglichen. Daher werden die Testfälle so implementiert, wie sie in den Szenarien beschrieben werden. Die Beschreibung der Szenarien folgt im nächsten Abschnitt.

Für die automatisierten Tests wird ein eigenes Projekt angelegt, in dem sich nur die Tests befinden. Für dieses Projekt wird auch ein eigenes privates Git-Repository auf Github angelegt. Die Tests sind ausschließlich für diese Untersuchung gedacht. Daher liegen sie nicht im Git-Repository der Oberfläche.

Die Tests werden mit dem Browser Firefox in der Version 55 ausgeführt. Dies betrifft sowohl die automatisierten Tests als auch die manuellen Tests.

Die Tests werden auf verschiedenen Betriebssystemen ausgeführt, je nachdem, was zur Verfügung steht. Es wird davon ausgegangen, dass das Betriebssystem keinen Einfluss auf die Aussagen der Untersuchungen haben wird.

4.3.2 Definitionen der Szenarien

Unter Beachtung der vorher genannten Punkte wurden verschiedene Szenarien entwickelt, die hier im einzelnen vorgestellt werden sollen. Dabei sind die einzelnen Szenarien nicht spezifisch

für das MARS Framework, sondern gelten allgemein für Microservice-basierte Systeme, die über eine Oberfläche bzw. Weboberfläche verfügen.

Für das Durchführen von Exploratory Testing ist ein ausreichender Kenntnisstand über die MARS Oberfläche vorhanden. Besonders das Importieren von Dateien ist bekannt, da bereits im Grund- und im Hauptprojekt mit dieser Funktion gearbeitet wurde [Pie17b, Pie17a]. Daher wird diese Funktion für die meisten Szenarien verwendet.

Es gilt dabei für alle Szenarien, dass Zeit als ein Kriterium genommen wird. Daher werden automatisierte sowie manuelle Tests daran verglichen, wie lange die Tests dauern. Dabei wird die Zeit für die Erstellung von Testfällen mit einbezogen. Das Testverfahren, welches am schnellsten abläuft, erfüllt das Kriterium.

Diese Kriterium ist wichtig, da an das Testkonzept für das MARS Framework definiert ist, dass es ein schnelles Feedback liefern soll. Aus diesem Grund spielt Zeit hier eine Rolle. Neben diesem Kriterium werden in den einzelnen Szenarien weitere Kriterien definiert.

Direkt im ersten Szenario wird Exploratory Testing mit einem bestehenden automatisierten Test verglichen. Der automatisierte Test ist dabei nach einem definierten Testfall implementiert. Dieser Testfall ist in der Tabelle 4.12 dargestellt. Dieser Testfall beschreibt das Importieren einer Datei über die Weboberfläche in das MARS Framework. Dieser explizite Testfall soll auch mit Exploratory Testing ausgeführt werden. Die beiden Ausführungen werden in der Dauer und dem Ergebnis der Ausführung verglichen. Im Programm wurden keine Änderungen vorgenommen. Deshalb wird angenommen, dass der automatisierte Test erfolgreich abläuft. Dieses Ergebnis wird auch beim manuellen Ausführen erwartet und bildet somit das zweite Kriterium.

Dieses Szenario steht durchaus im Widerspruch zu der Definition von Exploratory Testing in Kapitel 3.1.5. Exploratory Testing umfasst mehr als das Testen einer Funktion und vor allem umfasst es nicht das Ausführen eines expliziten Testfalls. Dennoch wird dieser Vergleich hier ausgeführt. Dies passiert aus folgendem Grund. Es soll untersucht werden, ob Exploratory Testing sinnvoll eingesetzt werden kann und ob es die bessere Alternative zu automatisierten Tests ist. Dazu ist es auch zwingend zu untersuchen, inwiefern Exploratory Testing im Bereich des Regressionstests Sinn ergibt. Denn genau das ist die Aufgabe des automatisierten Tests hier. Er soll aufgrund von Änderungen wiederholt prüfen, dass es keine Fehler gibt.

Im zweiten Szenario wird in der Oberfläche der Anwendung ein neues Feature hinzugefügt. In der Oberfläche wird eine neue Unterseite mit dem Titel “Help" angelegt, auf der Unterstützung angeboten werden soll. Auf der neuen Seite wird eine Tabelle mit verschiedenen Themen angezeigt, zu denen der Anwender Unterstützung bekommen kann. Dieses Feature wurde nur für dieses Szenario entworfen.

Tabelle 4.12: Testfall im ersten Szenario

Vorbedingung	Aktion	Erwartendes Ergebnis
Startseite wird angezeigt	• Im Menü “Data Management" die Option “Data Import" auswählen. • Felder mit Daten füllen. – Title -> “Test Title" – Tags -> “Tag,Tag,Tag" – Privacy -> “Public" – Description -> “This is a description" • Drücken des Buttons “Upload".	Es erscheint die Meldung “success".

Da es sich hier um ein neues Feature handelt, existiert für diese Funktion kein automatisierter Test. Mit diesem Szenario wird verglichen, wie Exploratory Testing und automatisierte Tests mit dem Testen von neuen Funktionen umgehen.

Wie am Anfang des Abschnitts erwähnt, wird auch hier das Kriterium Zeit gewählt. Neben der Quantität wird zudem die Qualität als Kriterium genommen. Damit ist gemeint, wie gut die neue Funktion getestet wird. Dazu gehört, ob alle funktionalen Anforderungen an die Funktion getestet werden und ob ggf. alle Fehler gefunden werden. Dazu wird auch der Prozess insgesamt beleuchtet, wie die Erstellung des Testfalls und die Implementierung beim automatisierten Test und der Prozess beim Exploratory Testing aussieht.

Bei dem dritten Szenario wird eine Änderung im HTML-Code der Oberfläche vorgenommen. Die Änderung hat keinen Einfluss auf die Darstellung der Oberfläche. Dafür wird auf der Seite, bei der die Daten für den Import eingegeben werden, die CSS-ID für die “Privacy" Einstellung verändert. Die ID wird von ´´privacy" auf “privac" geändert.

Die Änderung kann durch eine Fehlerbehebung ausgelöst werden, da z.B. eine falsche CSS-ID angegeben war und daher das Element nicht korrekt dargestellt wird.

Für den Vergleich wird wieder der automatisierte Test aus dem ersten Szenario ausgeführt, siehe Tabelle 4.12 . Beim Exploratory Testing wird in diesem Fall nicht der explizite Testfall

ausgeführt. Als Aufgabe für das manuelle Testen ist definiert, zu überprüfen, ob das Hochladen von Dateien funktioniert.

Zusätzlich zum Kriterium der Zeit wird in diesem Szenario das Kriterium der Stabilität und die Anpassung durch Änderungen eingeführt. Stabilität bedeutet hier, dass der Test trotz der Änderung abläuft und ein valides Ergebnis liefert. Das dritte Kriterium der Anpassung untersucht, wie groß der Aufwand ist, um den Test auf die Änderung anzupassen. Das Testverfahren, welches die geringste Anpassung an die Änderung benötigt, erfüllt dieses Kriterium.

Dazu gesellt sich noch das Kriterium des Ergebnisses. Besonders unter dem Punkt Stabilität reiht sich der Punkt Ergebnis ein. Wenn beide Tests stabil sind, sollten sie auch das gleiche Ergebnis liefern.

Das vierte und letzte Szenario beschreibt eine Änderung innerhalb des Services, welcher für die Oberfläche verantwortlich ist. Die Änderung hat weder Auswirkung auf den HTML-Code noch auf die dargestellte Oberfläche.

In diesem Szenario symbolisiert die Änderung eine Veränderung der Anforderung. Es wird die Methode angepasst, die die eingegeben Tags beim Import der Dateien aufbereitet, um sie an den Metadata-Service zu schicken.

Dafür wird für dieses Szenario in der Methode “convertTagsToArray" beim Datenimport eine Änderung vorgenommen. In der Methode werden die einzelnen Tags als String in ein Array geschrieben. Als Änderung wird vorgenommen. dass die Tags nicht als String sondern als Objekt übergeben werden. Dafür wird der Befehl “.text" beim Schreiben des Tags in den Arrary gelöscht. Dadurch enthält der Array die Tags nicht als Strings sondern als Objekte.

Da auch in diesem Szenario das Hochladen von Dateien verwendet wird, führt der automatisierte Test wieder den Testfall aus dem ersten Szenario aus (Tabelle 4.12). Wie im dritten Szenario wird die gleiche Aufgabe für Exploratory Testing definiert und zwar zu testen, ob das Feature Hochladen von Dateien funktioniert. Es wird kein expliziter Testfall für Exploratory Testing definiert, sondern in gleicher Herangehensweise, wie im dritten Szenario, vorgegangen.

Als Kriterium dient die Untersuchung des Testumfangs. Genauer gesagt wird untersucht, ob die geänderte Anforderung getestet wird bzw. ob die Tests registrieren, dass die Tags anders aufbereitet werden. Aus diesem Grund werden weder der automatisierte Test noch das Vorgehen bei Exploratory Testing angepasst. Wegen der fehlenden Anpassung müssten die Tests eigentlich fehlschlagen, da sich die Anforderung geändert hat. Durch einen Fehlschlag der Tests lässt sich besser aufzeigen, ob die Tests die Änderung überhaupt testen. Dazu wird auch wieder das Kriterium der Zeit genommen.

Aus Gründen der Übersicht sind die Szenarien mit ihren Kriterien in der nachfolgenden Tabelle 4.13 dargestellt.

Tabelle 4.13: Testsznenarien für die Untersuchung von Exploratory Testing

Nummer	Expliziter Testfall	Änderung	Beschreibung Änderung	Art der Änderung	Kriterium
1	Ja	Nein			Zeit & Ergebnis
2	Nein	Ja	neues Feature	neues Feature	Zeit & Qualität
3	Nein	Ja	Änderung im HTML-Code	Fehlerbehebung	Zeit & Stabilität & Anpassung & Ergebnis
4	Nein	Ja	Änderung im Services ohne Einfluss auf den HTML-Code und auf die gerenderte Oberfläche	Änderung einer Anforderung	Zeit & Testumfang

4.3.3 Durchführung der Szenarien

Im nachfolgenden Abschnitt werden die Ausführungen und die Ergebnisse der einzelnen Szenarios beschrieben. Dazu wird erläutert, ob Exploratory Testing die Kriterien erfüllen konnte.

Szenario 1

Zunächst wurde der automatisierte Test ausgeführt. Dafür wurde der Test in der Entwicklungsumgebung gestartet. Die Dauer des Tests war weniger als zwei Sekunden. Zudem war der Test erfolgreich, indem alle Felder gefüllt wurden und die Meldung “Success" erschien.

Danach wurde der Testfall manuell ausgeführt. Dabei wurden alle einzelnen Schritte, wie im Testfall eins definiert ausgeführt. Es wurden genau die Daten genommen und das Ergebnis erwartet, wie im ersten Testfall definiert war. Eine Variation davon fand nicht statt. Die Ausführung dauerte 30 Sekunden. Auch hier war der Test erfolgreich, wie beim automatisierten Test.

Beim Kriterium der Zeit war der manuelle Test deutlich langsamer als der automatisierte Test. Beim Ergebnis lieferten beide das gleiche Ergebnis ab, wie in Abbildung 4.9 dargestellt.

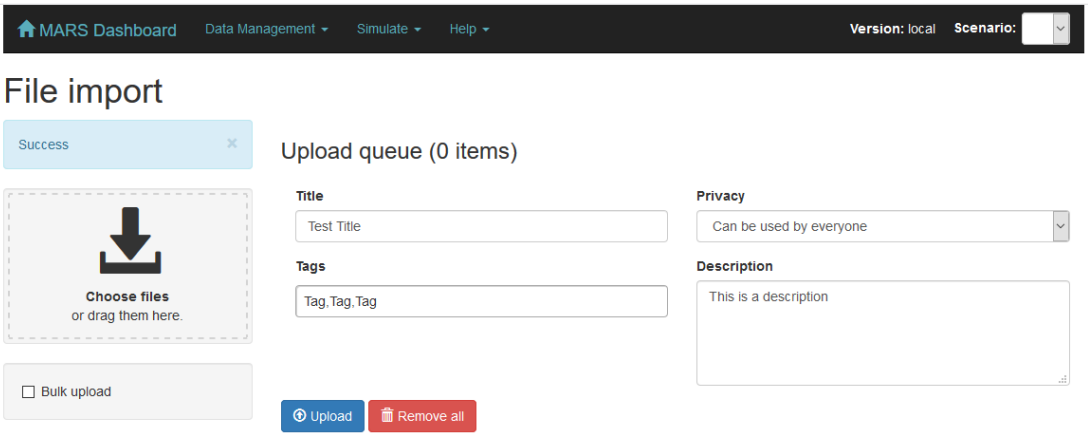

Abbildung 4.9: Erfolgreicher Testfall im ersten Szenario

Die Ergebnisse sind in der Tabelle 4.14 komprimiert. Dabei steht AT für automatisierter Test und ET für Exploratory Testing.

Tabelle 4.14: Ergebnisse im ersten Szenario

Testverfahren	Zeit	Kriterium Zeit	Kriterium Ergebnis
AT	circa 2s	Erfüllt	Erfüllt
ET	circa 30s	Nicht Erfüllt	Erfüllt

Szenario 2

Für den automatisierten Test musste zunächst ein Testfall definiert und im Anschluss implementiert werden. Für das neue Feature wurde der Testfall in der Tabelle 4.15 entworfen. Es soll die Seite “Help" aus dem Menü aufgerufen werden. Die Seite ist erfolgreich geladen, wenn eine Tabelle angezeigt wird. Auf Grundlage dieser Informationen wurde der Testfall mit Selenium implementiert.

Tabelle 4.15: Testfall im zweiten Szenario

Vorbedingung	Aktion	Erwartendes Ergebnis
Startseite wird angezeigt	Im Menü “Help" die Option “Help" auswählen.	Es wird eine Tabelle mit den Hilfsthemen angezeigt.

Für die Implementierung des Tests benötigt Selenium Selektoren, um auf die spezifischen Elemente zugreifen zu können. Daher wurden die einzelnen Schritte des Testfalls manuell

ausgeführt, um die Selektoren aus dem HTML-Code auszulesen. Danach wurde der Test ausgeführt, wobei der Test erfolgreich abgeschlossen wurde.

Das Entwerfen des Testfalls dauerte eine Minute, während die Implementierung fünf Minuten dauerte. Die Ausführung des Testfalls dauerte nur 700 Millisekunden.

Für Exploratory Testing wurde dahingehend nur definiert, dass die neue Funktion der “Help" Seite getestet werden soll. Dafür wurde die Oberfläche aufgerufen und im Menü die Option “Help" ausgewählt. Nach dem Laden der Seite wurde geprüft, ob die Tabelle vorhanden ist. Die Tabelle war vorhanden.

Zusätzlich wurde noch der Header der Seite geprüft. Diese Prüfung wurde dynamisch hinzugefügt, da der Header auf den ersten Blick anders aussah als bei den anderen Seiten, wie z.B. bei der Seite “Import" (Abbildung 4.10 und Abbildung 4.11). Dabei wurde festgestellt, dass der Button, der dazu dient, um auf die Startseite zurück zu gelangen, nicht funktionierte. Dazu wurden auch die Version der Anwendung und die anderen Seiten nicht angezeigt. Da dies auf den anderen Seiten funktionierte, wurde dies als Fehler interpretiert, so dass das neue Feature nicht als fehlerfrei implementiert betrachtet werden kann. Daher war diese Ausführung nicht erfolgreich.

Abbildung 4.10: Header auf der Seite Help

Abbildung 4.11: Header auf der Startseite

Die manuelle Ausführung dauerte nur ein paar Sekunden. Dadurch war diese deutlich schneller als der automatisierte Test. Dabei benötigen beim automatisierten Test besonders das Entwerfen und die Implementierung relativ viel Zeit. Bei der reinen Ausführung war der automatisierte Test wieder schneller.

Auch beim zweiten Kriterium der Qualität liegt Exploratory Testing vor dem automatisierten Test. Beim automatisierten Test wurde der Fehler mit dem Header nicht gefunden, während bei der manuellen Ausführung dieser festgestellt wurde. Der Fehler wurde nicht sofort festgestellt, sondern es musste erst verglichen werden, wie sich der Header auf den anderen Seiten verhält.

Der Vorteil von Exploratory Testing hierbei ist, dass während der Ausführung des Tests gelernt werden kann. Es wurde gelernt, dass der Header einen Button für das Laden der

Startseite und die Versionsnummer enthält. Auf Grundlage dieses neuen Wissens konnte der Fehler entdeckt werden.

Dazu wurde das Wissen gewonnen, dass der implementierte Testfall für das zweite Szenario unzureichend ist, da er diesen Fehler nicht entdeckte. Mithilfe der Informationen, die vom Ausführen mit Exploratory Testing stammen, kann der automatisierte Testfall im zweiten Szenario aktualisiert werden, damit dieser zukünftig auch diesen Fehler entdeckt.

Die Ergebnisse sind in der Tabelle 4.16 zusammengefasst.

Tabelle 4.16: Ergebnisse im zweiten Szenario

Testverfahren	Zeit	Kriterium Zeit	Kriterium Qualität
AT	circa 300s	Nicht Erfüllt	Nicht Erfüllt
ET	circa 10s	Erfüllt	Erfüllt

Szenario 3

Der Testfall war beim automatisierten Test nicht erfolgreich. Beim Setzen des Wertes bei “privacy" wurde die Exception “NoSuchElementException" geworfen.

Die Exception stammt von Selenium und sagt aus, dass das gesuchte Element mit dem gegebenen Selektor nicht gefunden wurde. Im implementierten Testfall wurde nach dem Element mit der CSS-ID “privacy" gesucht. Da nach der Änderung das Element nun die CSS-ID “privac" besitzt, wurde das Element nicht gefunden und die beschriebene Exception erschien.

Die Ausführung des automatisierten Tests dauerte circa 10,9 Sekunden. Die Dauer ist vornehmlich dadurch bedingt, dass ein Timeout von zehn Sekunden im Test eingestellt ist, bis das gesuchte Element gefunden sein soll. Der Timeout hat die Aufgabe, längere Ladezeiten von Seiten und Ähnliches zu kompensieren, damit Testfälle nicht fehlschlagen, wenn ein Element nicht sofort zur Verfügung steht.

Für die Ausführung bei Exploratory Testing wurde die Seite “Import" aus dem Menü ausgewählt und geladen. Darauf wurde geprüft, ob alle Felder vorhanden sind und ob sie gefüllt werden können. Dabei wurde auf die Erfahrungen zurückgegriffen, wie die Seite in den bisher ausgeführten Tests aussah, denn es gab keine Informationen darüber, dass die Seite anders aussehen soll. Aufgrund dieser Kenntnisse wurden die Felder mit validen Daten gefüllt und der Button für den Import wurde gedrückt. Nach dem die Meldung “Sucess" erschien, wurde noch der Header geprüft. Diese Überprüfung entstand aus der Erfahrung aus dem zweiten Szenario. Neben der Prüfung des Vorhandenseins der Versionsnummer wurde der Button, um die Startseite zu laden, ausgelöst. Nachdem die Startseite geladen worden ist, wurde der Test

beendet und als erfolgreich vermerkt. Eine Veränderung des Selektors des Elements “Privacy" wurde nicht wahrgenommen.

Die Ausführung dauerte dabei circa 35 Sekunden. Beim Kriterium der Zeit ist der automatisierte Test schneller und zwar circa dreimal so schnell.

Jedoch schlägt der automatisierte Test aufgrund einer Änderung fehl, die beim Exploratory Testing keine Auswirkung hat. Dadurch wirkt Exploratory Testing stabiler als der automatisierte Tests mit Selenium. Die Änderung am Selektor wurde bewusst vorgenommen, um aufzuzeigen, dass Tests mit Selenium wegen dieser vermeintlichen kleinen Änderung schnell fehlschlagen können.

Dabei dienen die Selektoren nur als Unterstützung für den automatisierten Test. Zwar wird damit auch überprüft, ob das entsprechende Element vorhanden ist, die Kernaufgabe liegt jedoch darin, dass Element für den Test zugänglich zu machen.

Durch die Exception liegt die Vermutung nahe, dass ein Fehler gefunden wurde und zwar der Fehler, dass das Element nicht vorhanden ist. Trotzdem liegt kein Fehler vor, denn das Element ist ja vorhanden, es hat sich nur der Selektor geändert. Daher muss bei Änderungen, die ggf. auch die Tests betreffen können, geprüft werden, ob die Tests aktualisiert werden müssen. In diesem Szenario bedeutet dies, dass nach der Änderung im HTML-Code auch der Selektor im Test angepasst werden müsste. Daher geht der Punkt bei diesem Kriterium ans Exploratory Testing, da bei diesem Testverfahren keine Änderung notwendig war.

Wegen dieser nicht vorhandenen Änderung, kommt beim automatisierten Test auch ein anderes Ergebnis heraus. Die Änderung betrifft weder die dargestellte Oberfläche noch die Funktion. Daher sollte der Tests erfolgreich sein, so wie er beim Exploratory Testing war. Aus diesem Grund schneidet Exploratoryy Testing auch beim Kriterium Ergebnis in diesem Fall besser ab.

Eine Zusammenfassung der Ergebnisse ist in der Tabelle 4.17 dargestellt.

Tabelle 4.17: Ergebnisse im dritten Szenario

Testverfahren	Zeit	Kriterium Zeit	Kriterium Stabilität	Kriterium Anpassung	Kriterium Ergebnis
AT	circa 10s	Erfüllt	Nicht Erfüllt	Nicht Erfüllt	Nicht Erfüllt
ET	circa 35s	Nicht Erfüllt	Erfüllt	Erfüllt	Erfüllt

Szenario 4

Die Durchführung ist hier sehr ähnlich wie beim dritten Szenario. Im Gegensatz zum dritten Szenario ist hier der automatisierte Test wieder erfolgreich und läuft in wenigen Sekunden ab.

Dazu wurden bei Exploratory Testing die gleichen Schritte gewählt wie im dritten Szenario. Daher ist auch die Dauer der Ausführung mit 45 Sekunden ähnlich. Auch hier war die Durchführung erfolgreich.

Damit liegt der automatisierte Test beim Kriterium Zeit wieder vorne. Beim zweiten Kriterium, dem Entdecken bzw. Testen der Änderung, liefern beide das gleiche Ergebnis. Weder beim automatisierten Test noch bei Exploratory Testing wurde die Änderung festgestellt, denn beide Tests waren erfolgreich.

Bei der Ausführung von Exploratory Testing wurden noch zusätzlich die Entwicklertools von Firefox hinzugenommen. Selbst damit konnte die Änderung nicht entdeckt werden, da es keine Fehlermeldung oder ähnliches in der Web-Konsole des Browsers gab. Aufgrund der Verwendung dieser Tools erhöhte sich zudem die Dauer

Dass beim automatisierten Test die Änderung nicht entdeckt worden ist, war keine Überraschung. Denn der automatisierte GUI-Test kontrolliert nur, ob die Oberfläche funktioniert. Methoden und Funktionen im Hintergrund werden nicht mitgetestet. Es ging in diesem Fall auch eher darum, zu prüfen, ob mit Exploratory Testing ein größerer Testumfang existiert. Dies kann nicht bestätigt werden.

Für die Übersicht sind die Ergebnisse in der Tabelle 4.18 zusammengetragen.

Tabelle 4.18: Ergebnisse im vierten Szenario

Testverfahren	Zeit	Kriterium Zeit	Kriterium Testumfang
AT	circa 2s	Erfüllt	Nicht Erfüllt
ET	circa 45s	Nicht Erfüllt	Nicht Erfüllt

4.3.4 Bewertung der Ergebnisse

Aufgrund der verschiedenen Ergebnisse können verschiedene Aussagen getroffen werden. Zusätzlich kann bezüglich der Implementierung von Systemtests mit Selenium auf die Erkenntnisse aus dem Grundprojekt zurückgegriffen werden. Die gesammelten Erfahrungen werden hier mit einbezogen [Pie17b].

Die Frage nach automatisierten Tests oder Exploratory Testing ist keine Entweder-Oder-Frage. Beide Testarten haben in bestimmten Situationen ihre Existenzberechtigung und stehen nicht in Konkurrenz zu einander.

Automatisierte Tests bieten sich vor allem bei Regressionstests an. Besonders die deutlich geringere Ausführungsgeschwindigkeit ist der Hauptgrund dafür, denn dadurch können mehr Testfälle ablaufen. Nur bei dem zweiten Szenario war das Testen mit Exploratory Testing schneller.

Das Entwerfen und Implementieren von automatisierten Tests kann durchaus Zeit kosten. Dabei ist die längere Vorbereitungszeit kein Problem, da sie nur einmal investiert werden muss. Nachdem der Test implementiert worden ist, kann er immer wieder ausgeführt werden. Dadurch rentiert sich die längere Vorbereitungszeit ab einem bestimmten Punkt. Dennoch ist schon ab dem zweiten Durchlauf eine deutliche Verbesserung zu spüren, da nicht zehn Sekunden auf ein Ergebnis gewartet werden muss, sondern weniger als eine Sekunde. Dazu verringert sich der Aufwand, da der Computer das Testen übernimmt.

Wie im dritten Szenario veranschaulicht, können automatisierte Tests mit einem Fehler enden, obwohl kein Fehler vorliegt. Das im Szenario verwendete Framework Selenium ist darauf angewiesen, über bestimmte Selektoren Elemente zu finden. Ohne dies ist ein automatisiertes Testen nicht möglich. Dadurch können automatisierte Tests sehr schnell fehlschlagen und dies macht sie insgesamt instabiler als manuelle Tests. Bei manuellen Tests fallen ggf. solche Änderungen nicht auf oder sie können durch menschliche Intuition umgangen werden, so dass der Test weiter ablaufen kann. Dazu zeigt das dritte Szenario auf, dass die automatisierten Tests gewartet werden müssen. Bei Änderung von Selektoren oder Ähnlichem, was Auswirkungen auf die Tests haben kann, müssen auch die Tests aktualisiert werden.

Ein weiterer Aspekt der automatisierte Tests instabiler machen kann, ist die Verwendung von asynchronen Aufrufen. Viele Webseiten verwenden asynchrone Aufrufe. Dies erschwert das Testen dahingehend, da sich das Verhalten bei jedem Durchlauf eines Tests unterscheiden kann. Die Entwicklung von stabilen Tests, die damit umgehen können, ist durchaus zeitintensiv und komplex. Je nach dem wie komplex die Webseite ist und wie viele asynchrone Aufrufe sie enthält, umso mehr muss bei der Testentwicklung/Testdurchführung darauf geachtet werden.

Die zusätzliche Implementierung von Sicherheitsmechanismen, damit asynchrone Aufrufe den Testablauf nicht stören, verlängert nicht nur die Implementierung des Tests, sondern kann auch dessen Ablauf verlängern. Diese Sicherheitsmechanismen bestehen meistens aus Methoden, die eine definierte Zeit festschreiben bis eine bestimmte Aktion eintritt oder das gesuchte Element erscheint. Wie im dritten Szenario wird zehn Sekunden gewartet, bis der Fehler geworfen wird, dass das Element nicht vorhanden ist. Da in diesem Fall das Element

mit dem gegebenen Element nie gefunden werden kann, wirkt diese Zeit als sinnlos eingesetzt. Jedoch kann nie sicher gesagt werden, ob das Element immer sofort geladen wird oder bei manchen Testabläufen etwas mehr Zeit benötigt wird. Dahingehend ist dieser Mechanismus notwendig, um die Stabilität des Tests zu erhöhen.

Anderseits können instabile Tests die Folge haben, dass das Vertrauen in die Tests sinkt, und es für den Normalfall erachtet wird, dass der Test fehlschlägt. Dieser Umstand wurde bereits im Abschnitt zur Vorstellung des Testkonzeptes von Newman beschrieben, er bezieht sich auf Vaughan und den Begriff des “normalization of deviance" [Vau97].

Dennoch ist es schwierig, Tests zu implementieren, die nicht wegen unterschiedlichen Verhaltens, bedingt durch asynchrone Aufrufe, fehlschlagen. Dazu kann auch schon die Auswahl der Selektoren für die Elemente herausfordernd sein, damit die Elemente auch immer sicher gefunden werden.

Eine weitere Herausforderung ist, die Tests so zu entwerfen, dass bei einem Fehler die Meldung den Fehler auch explizit benennt. Sollte dies nicht der Fall sein, entsteht teilweise die Situation, den Testfall erneut manuell auszuführen, um den Fehler zu finden und detaillierter beschreiben zu können. Dies wiederum steht im Widerspruch zum Nutzen des automatisierten Tests.

Trotz dieser Schwierigkeiten bieten sich automatisierte Tests am besten für Regressionstests an. Dazu ist die Ausführung der immer gleichen Testfälle sehr monoton. Daher ergibt es keinen Sinn, diese Tests durch einen Menschen ausführen zu lassen, weil die Vorteile nicht genutzt werden.

Da manuelle Tests von einem Menschen ausgeführt werden, sollten in den Anwendungsfällen auch die spezifischen Vorteile, z.B. der individuellen Reaktionsmöglichkeiten auf Problemlagen, genutzt werden, da der Mensch mehr Ressourcen verbraucht als ein Computer. Beim Exploratory Testing können diese Vorteile genutzt werden, wie das zweite Szenario zeigt. Denn hier konnte der Fehler aufgrund der menschlichen Intuition gefunden werden.

Das nicht korrekte Verhalten im Header ist nur beim Testen mit Exploratory Testing aufgefallen. Der automatisierte Test hatte keine Prüfung für den Header implementiert, deshalb konnte er keine Aussage zum nicht korrekten Verhalten treffen. Bei der Ausführung mit Exploratory Testing konnte dagegen dynamisch auf die Situation reagiert und geprüft werden, wie der Header auf den anderen Seiten aussieht. Darüber hinaus war es möglich diesen Fehler zu interpretieren.

Generell bietet sich Exploratory Testing immer für das Testen neuer Funktionen an. Bei neuen Funktionen existieren meistens noch keine automatisierten Testfälle. Deshalb sollten neue Funktionen intensiver getestet werden, denn dadurch können wichtige Informationen

gesammelt werden, die für die Implementierung eines automatisierten Testfalls benötigt werden. Der Mensch kann sich schneller und intuitiver auf die neue Funktion einstellen. Dazu kann auf Erfahrungen zurückgegriffen werden, um ggf. typische Fehler, die bei einer Funktion auftreten können, zu testen. Ergänzend dazu kann während des Testens entschieden werden, ob bestimmte Auffälligkeiten Fehler sind. Wie im zweiten Szenario beschrieben, können zusätzliche Informationen gesammelt werden, mit denen diese Entscheidung getroffen werden kann.

Dazu benötigt Selenium Selektoren, um die Elemente zu finden. Diese Informationen werden meist mit einer manuellen Ausführung des zu implementierten Tests gewonnen. Wird dieser Schritt mit in das Exploratory Testing einbezogen, muss dieser Schritt nicht erneut ausgeführt werden.

Es existieren dafür verschiedene Werkzeuge, mit denen die Tests aufgezeichnet werden können. Auf Grundlage dieser Aufzeichnungen und der dadurch gesammelten Informationen können automatisierte Testfälle entstehen.

Diesen Ansatz verfolgt auch Google, wie bereits in der Vorstellung des Testkonzeptes von Google erwähnt wurde. Jeder manuelle Test wird hierbei aufgezeichnet und auf Grundlage dessen wird der automatisierte Test implementiert.

Zum Schluss zeigt das Ergebnis aus dem vierten Szenario auf, dass manuelle und automatisierte GUI-Tests nicht alle Fehler finden können. Dahingehend ist es unumgänglich die Oberfläche auch auf anderen Ebenen zu testen, um diese Fehler zu finden. Besonders moderne JavaScript-Frameworks bringen ein MVC-Pattern mit sich. Dadurch enthalten sie häufig komplexeren Sourcecode, der neben der Beschreibung der Oberfläche auch Logik beinhaltet.

Zusammenfassend für dieses Kapitels gilt, dass sich automatisierte Tests trotz ihrer Nachteile bezüglich der Stabilität am besten für Regressionstest eignen, da sie aufgrund der deutlich schnelleren Ausführungsgeschwindigkeit effizienter sind. Exploratory Testing bietet sich vor allem bei neuen Funktionen und als Grundlage für automatisierte Tests an. Besonders bei neuen Funktionen macht es Sinn, diese intensiv zu testen, denn dadurch kann die Basis für automatisierte Tests erarbeitet werden. Dennoch muss beachtet werden, dass bei komplexen Oberflächen reine GUI-Tests nicht ausreichen, sondern weitere Tests auf anderen Ebenen erforderlich sind.

4.4 Fazit

In diesem Kapitel wurden die vielfältigen Herausforderungen ausgearbeitet, die beim Testen einer Microservice-basierten Anwendung auftreten. Viele der Herausforderungen ähneln den Herausforderungen beim Testen einer verteilten Anwendung.

Dazu ergänzend wurden die Herausforderungen aufgezählt, die durch das MARS Framework begründet sind. Neben diesen Herausforderungen gelten besonders die Anforderungen beim Testen, die durch den ständigen Wandel der Microservices und der gesamten Anwendung entstehen. Dafür sind die verschiedenen Arten von Änderungen definiert. Auf diese verschiedenen Arten von Änderungen muss das Testkonzept eine Lösung präsentieren.

Für die Entwicklung eines Testkonzeptes für das MARS Framework sind mehrere verschiedene Testkonzepte untersucht wurden. Dabei sind nicht alle explizit für Anwendungen mit Microservices gedacht. Auffallend bei der Untersuchung der Testkonzepte ist, dass teilweise große Unterschiede zwischen den Konzepten zu finden sind, obwohl der größte Teil davon explizit für Microservices gedacht ist. Dabei hat jedes Konzept seine Vor- und Nachteile. Die daraus gewonnenen Aspekte werden für die Entwicklung eines Testkonzeptes für das MARS Framework wieder verwendet.

Ergänzend dazu wurde in diesem Kapitel untersucht, inwiefern Exploratory Testing in einer Microservice-basierten Anwendung Sinn ergibt. Grundsätzlich ist festzustellen, dass Exploratory Testing in bestimmten Testbereichen, wie beim Testen von neuen Anforderungen, große Vorteile bietet.

5 Entwicklung eines Testkonzeptes für das MARS Framework

In diesem Kapitel wird das Testkonzept für das MARS Framework vorgestellt und erläutert. Dafür werden zuerst die Anforderungen aufgezählt. Im Anschluss daran wird das Testkonzept vorgestellt. Danach werden die einzelnen Ebenen und die Art und Weise, wie das Testkonzept mit den verschiedenen Arten von Änderungen umgeht, erläutert. Darauf werden verschiedene Ausblicke für bestimmte Anwendungsfälle gewährt, wie z.B. das Testkonzept für ein allgemeines Microservice-basiertes System angepasst werden muss. Zum Abschluss dieses Kapitels wird noch ein Fazit über das erstellte Testkonzept gezogen.

5.1 Anforderungen

Das Testkonzept für MARS soll verschiedene Anforderungen erfüllen. Dazu gehört, dass das Testkonzept auf die gesammelten Herausforderungen (Abschnitt 4.1) Lösungen bereitstellt. Des Weiteren muss das Testkonzept wiedergeben, wie es mit den verschiedenen Arten von Änderungen umgehen möchte. Zusätzlich darf das Testen nicht zu einem Flaschenhals in der Deployment Pipeline führen.

Bei allen Aspekten muss beachtet werden, dass das Testkonzept in ein bestehendes System eingegliedert werden kann.

5.2 Testkonzept

In diesem Abschnitt wird das entwickelte Testkonzept für das MARS Framework präsentiert. Die Vorstellung erfolgt ohne ausführliche Begründung, weil diese wird im nachfolgenden Abschnitt 5.3 gegeben. Für die Erstellung des Testkonzeptes wurden Einschränkungen vorgenommen.

Das Testkonzept beschäftigt sich ausschließlich mit dem Testen von funktionalen Anforderungen. Für das MARS Framework besteht zum Zeitpunkt dieser Arbeit kein Testkonzept. In diesem Zusammenhang wurde entschieden, sich zuerst auf die funktionalen Anforderungen zu

konzentrieren. Das Testen von funktionalen Anforderungen wird als bedeutsamer eingestuft als das Testen der nicht-funktionalen Anforderungen.

Services, die ausschließlich bei der Durchführung von Simulationen Verwendung finden, werden von diesem Testkonzept nicht abgedeckt. Dafür existieren mehrere Gründe. Zu einem gibt es bisher keine Lösung, diese Services ohne den Aufwand einer Simulation zu testen. Zum anderen fehlt auch das Wissen über die genaue Funktionsweise dieser Services. Dazu existiert die Schwierigkeit, wie in den MARS spezifischen Herausforderungen beschrieben, dass die Anforderungen für diesen Bereich abhängig von der Simulation sind. Dahingehend ist es schwierig, ein Testkonzept zu entwerfen, da die Anforderungen nicht bekannt sind.

Das Testkonzept beruht auf dem Wissen über die Services, die das Importieren von Dateien verwalten. Die daraus gewonnenen Schlussfolgerungen wurden auf die anderen Services projektiert. Das hier vorgestellte Testkonzept gilt für alle Microservices im MARS Framework, außer auf die eben gerade beschriebene Ausnahme.

5.2.1 Allgemein

Ähnlich wie beim Testkonzept von Wolff ist das Testkonzept unterteilt. Es gibt ein Testkonzept für jeden Microservice und ein Testkonzept für das Gesamtsystem. Die beiden Testkonzepte sind jedoch eng miteinander verbunden. Nur in der Kombination der beiden Testkonzepte können die Anforderungen umgesetzt werden.

Der Grund dafür ist, dass die beiden Testkonzepte aufeinander aufbauen. Das Testkonzept für das Gesamtsystem benötigt als Grundlage das Testkonzept für einen einzelnen Microservice. Es ergibt erst Sinn das Gesamtsystem zu testen, wenn der Microservice seine Tests bestanden hat. Ohne bestandene Tests kann nicht sichergestellt werden, dass der Microservice seine Anforderungen erfüllt. Dadurch kann das Gesamtsystem ggf. auch seine Anforderungen nicht erfüllen. Daher wird zuerst das Testen eines einzelnen Microservice stattfinden und im Anschluss daran das Testen des Gesamtsystems.

Basierend auf dieser Grundlage wird zuerst das Testkonzept für einen einzelnen Microservice vorgestellt und danach das für das Gesamtsystem. Als Beispiel für die Erklärungen werden die Services aus dem Import-Bereich verwendet.

5.2.2 Testkonzept Microservice

Das Testkonzept für einen einzelnen Microservice ist in Abbildung 5.1 dargestellt. Auch hier wird das Testkonzept in Form einer Pyramide veranschaulicht. Das Fundament bilden die

Unit-Tests. Darauf aufbauend kommen die Integrationstests. Auf diese Ebene folgen die Service Tests. Zum Schluss folgen mögliche manuelle Tests in Form von Exploratory Testing.

Es muss für jeden Microservice einzeln definiert werden, wie die Tests explizit aussehen und in welcher Intensität der Microservice getestet werden soll. Schließlich ist jeder Microservice ein eigenständiges Programm. Dies muss sich im Testen widerspiegeln.

Ein Beispiel aus dem Import-Bereich verdeutlicht dies. Der Microservice “FileService" verhält sich ähnlich wie ein Proxy. Er prüft die Anfragen und leitet sie an den passenden Microservice weiter, der den eigentlichen Import ausführt. Aufgrund dieses Verhaltens kann es Sinn ergeben, den “FileService" mit einem anderen Fokus zu testen als die drei Microservices, die Importieren. So könnten z.B. beim “FileService" die Schnittstellen deutlicher im Fokus stehen und bei den anderen Microservices eher Integrationstests, um die Datenbankanbindung zu testen. Trotzdem existieren allgemeingültige Regeln für alle Microservices.

Im Folgenden wird der Aufbau der Pyramide genauer erläutert. Die Vorstellungen der einzelnen Testebenen folgen dabei dem Muster, dass zuerst die Definition genannt wird. Danach wird die Ebene ausführlicher erläutert und die Vorstellung endet mit meinem Beispiel.

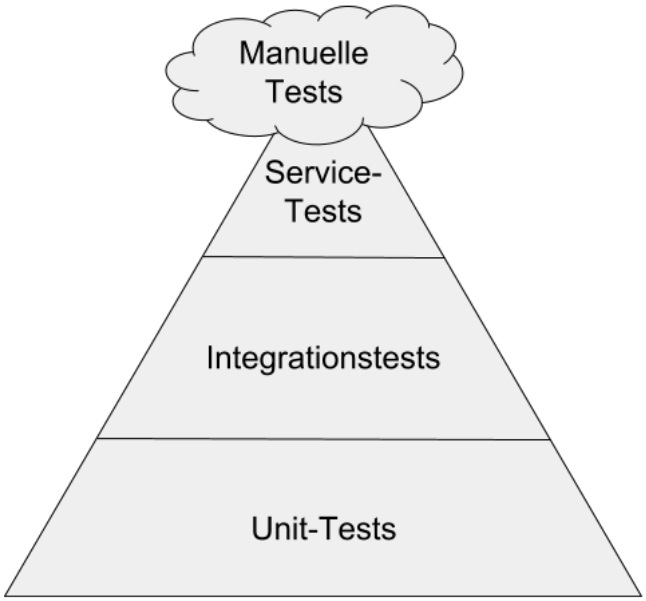

Abbildung 5.1: Testpyramide für einen Microservice im MARS Framework

Unit-Tests

Die Unit-Tests testen die kleinste Einheit im Sourcecode. Daher haben sie die Aufgabe eine einzelne Funktion oder Methode zu testen. Die Unit-Tests bilden die schnellste Testart von allen Testarten.

Besonders bei Änderungen am Sourcecode müssen Unit-Tests entworfen und implementiert werden. Jede Änderung am Sourcecode muss mit Unit-Tests abgesichert werden. Hilfreich ist dabei den Ansatz Test-Driven-Development anzuwenden.

Besonders wichtig ist, dass redundante Tests vermieden werden. Sollten bestimmte Module mehrmals verwendet werden, ist es nur einmal vonnöten Tests dafür zu entwerfen. Dieser Hinweis bezieht sich insbesondere auf den Metadata-Client. Es genügt, wenn an einer Stelle Unit-Tests für dieses Objekt entworfen werden. Die Tests werden dann bei jedem Erstellen eines Microservices, die dieses Objekt verwenden, mit ausgeführt.

Dahingehend ist es auch wichtig, feste Zuständigkeiten für die Tests zu benennen. Grundsätzlich sind immer die Personen dafür zu ständig, die die Änderung am Sourcecode vorgenommen haben. Ändert z.B. Entwickler A eine Methode, ist dieser Entwickler auch dafür verantwortlich, den dazugehörigen Test zu entwerfen und zu schreiben, der seine Änderung abdeckt.

Die Unit-Tests müssen komplett automatisiert und isoliert ablaufen. Daher müssen Technologien eingesetzt werden, die die Erstellung von automatisierten Tests unterstützen. Als Beispiel im Umfeld von Java können Frameworks wie JUnit für die automatisierten Tests und Mockito für die Erstellung von Mocks verwendet werden. Wichtig ist dabei zu klären, welche Technologien für die vorliegende Programmiersprache existieren. Deshalb muss eine Entscheidung getroffen werden, welche Technologien eingesetzt werden sollen. Die Entscheidung muss von dem Entwickler bzw. Team verantwortet werden, die den Microservices entwickeln.

Ergänzend dazu müssen die Tests eine fest definierte Höchstdauer besitzen. Für das MARS Framework ist eine Höchstdauer von einer Sekunde pro Unit-Test sinnvoll. Nach dem Entwerfen der Unit-Tests muss geprüft werden, ob die Tests diese Grenze erfüllen. Wenn dies nicht der Fall sein sollte, muss geprüft werden, woran es liegt. Zu einem kann der Unit-Test inperformant entworfen sein oder zum anderen kann die zu testende Methode zu umfangreich oder selber inperformant sein. Im Zuge dessen muss entweder der Test angepasst werden oder es wird diskutiert, die Methode zu überarbeiten und ggf. aufzuteilen. Sollte eine Überarbeitung der Methode keinen Sinn ergeben, muss die definierte Höchstdauer in Frage gestellt werden.

Im nachfolgende Code-Beispiel 5.1 ist ein Unit-Test abgebildet, der im Metadata-Client Objekt die Methode “getInstanceInfoForServiceName()" testet. Da ausschließlich diese Methode getestet werden soll, wird vom EurekaClient ein Mock verwendet, weil innerhalb der zu testenden Methode Funktionen vom EurekaClient aufgerufen werden.

```
@Test
public void testGetInstanceInfoForServiceName(){
    EurekaClientMock mockEureka = new EurekaClientMock();
    EurekaClientWrapper wrapper
    = new EurekaClientWrapper(mockEureka.getMockEurekaClient());

    InstanceInfo instanceInfo
    = wrapper.getInstanceInfoForServiceName("metadata");

    Assert.assertNotNull(instanceInfo);
}
```

Listing 5.1: Beispiel Unit-Test

Integrationstests

Bei den Integrationstests wird die Zusammenarbeit innerhalb eines Microservices getestet. Dabei sollen zu einem das Zusammenwirken von mehreren Methoden getestet werden. Zum anderen soll auch der Sourcecode getestet werden, welcher die Kommunikation mit externen Datenspeichern und externen Services übernimmt. Das Testen, ob die Zusammenarbeit mit anderen Services funktioniert, wird nicht getestet.

Zuerst muss in den jeweiligen Microservices geschaut werden, bei welchen Modulen sich Integrationstests auszahlen. Nicht jede Zusammenarbeit von Methoden ist lohnenswert zu testen. Integrationstests ergeben dort Sinn, wo komplexe Module bzw. Methoden miteinander agieren. Eine Methode A die z.B. die Methode B und C aufruft, welche beide einen Boolean-Wert zurückgeben, mit denen via UND-Verknüpfung der Rückgabewert von A bestimmt wird, lohnt sich nicht zu testen, denn die vorliegende Logik in der Zusammenarbeit ist nicht komplex genug. Stattdessen sollten diese Methoden via Unit-Tests getestet werden.

Module, die die Kommunikation mit externen Datenspeichern oder Services übernehmen, rechnen sich immer für einen Integrationstest. Bei beiden Modulen werden wichtige Aufgaben übernommen, die grundlegend für die Funktion des Microservices sind.

Die von Clemson beschriebene Unterteilung in “Persistence Integration-Tests" und in “Gateway Integration-Test" ist dabei sehr hilfreich, diese Module zu testen [Cle14].

Mit “Persistence Integration-Tests" werden die Module für die Datenbankanbindung überprüft. Der Fokus beim Testen liegt darauf, das Mapping, wenn ein OR-Mapper verwendet wird, und das Verhalten bei typischen Fehlern, wie z.B. Timeouts, zu prüfen. Die Datenbank kann dabei vollständig simuliert werden. Es ist jedoch auch möglich, nur beim Prüfen des Verhaltens

bei typischen Fehlern die Datenbank zu simulieren und beim Testen des Mappings eine reelle Datenbank zu verwenden.

Beim Testen des Verhaltens bei Fehlern wird eine Testkopie von der Datenbank benötigt, damit die Fehler reproduzierbar erzeugt werden können. Genauso ist eine Testkopie vom Vorteil, wenn auf bestimmte Werte geprüft werden soll. Mit der Testkopie können die Testdaten besser verwaltet werden. Anders sieht es beim Testen des Mappings aus. Es wird hierbei nicht auf explizite Werte geprüft, sondern ob Name und Typ, wie im Mapping definiert, mit der Datenbank übereinstimmen. Daher wird nicht unbedingt eine Testkopie benötigt. Die Entscheidung ist davon abhängig, wie komplex die Erstellung einer Testkopie für die Datenbank ist und wie schnell die reelle Datenbank antwortet.

Beim “Gateway Integration-Test" werden die Module getestet, die die Kommunikation mit dem Netzwerk verwalten. Auch hier ergibt es Sinn eine Testkopie zu verwenden. Bei diesen Tests stellt das Modul Anfragen, die von einer Testkopie beantwortet werden. Die Testkopie antwortet darauf mit typischen Fehlern, wie z.B. falschen Headern oder falschen Bodys. Auch das Verhalten bei Timeouts kann hier getestet werden.

Module, die nicht getestet werden sollen, müssen simuliert werden. Dies betrifft z.B. externe Datenspeicher oder Module, die nicht im Fokus des Tests stehen. Bei der Erstellung der Testkopien werden Stubs bevorzugt.

Ähnlich wie bei den Unit-Tests müssen auch für die Integrationstests feste Zuständigkeiten definiert werden. Dafür muss unter den beteiligten Entwicklern abgesprochen werden, wer für welche Tests zuständig ist. Ergänzend dazu wird auch hier eine absolute Höchstdauer definiert. Als Orientierung kann eine Höchstdauer von 10 Sekunden pro Test gelten.

Im nachfolgenden Listing 5.2 ist eine Methode abgebildet, die als Beispiel für einen Integrationstests genommen werden soll. In der Methode wird über die “dataId" versucht, ein Metadata-Objekt aus der Datenbank zu laden. Diese Methode befindet sich im Microservice Metadata.

Bestimmte Technologien, wie z.B. “jOOQ" , bieten an, einen Stub von einer Datenbank zu erstellen, damit keine reellen Zugriffe auf der Datenbank erfolgen müssen. Dazu muss nicht sichergestellt werden, dass Daten mit der spezifischen dataId vorliegen. Der Stub kann so konfiguriert werden, dass er auf diese Anfrage den entsprechenden Wert zurückliefert.

Neben dem Stuben der Datenbank bietet sich auch an die Methode initDb() mit z.B. Mockito zu stuben. Das Initialisieren der Datenbank soll hier nicht getestet werden. Dafür würde sich ein eigener Testfall lohnen. An dieser Stelle soll das Lesen aus der Datenbank getestet werden.

Der Integrationstest kann ähnlich wie bei den Unit-Tests mit JUnit geschrieben werden. Unter Beachtung der Hinweise, die Clemson zu “Persistence Integration-Tests" gibt, wäre es

auch sinnvoll zu testen, wenn die Methode “getCollection" Null zurück gibt. Dadurch wird der Fall getestet, wenn die Datenbank nicht gefunden wird.

```
public Metadata getMetadata(String dataId)
throws DataIdUnknownException {
    initDb();

    DBCollection collection = db.getCollection(META_DATA_COLLECTION);
    DBObject metadata
    = collection.findOne(new BasicDBObject("dataId", dataId));
    if (metadata == null){
        throw new DataIdUnknownException
        ("There was no meta data for the given import id: "
         + dataId);
        }
    // The property "_id" is a mongo db specific key,
    // which we don't want to show the user
    metadata.removeField("_id");

    return new Gson().fromJson(metadata.toString(), Metadata.class);
}
```

Listing 5.2: Beispielmethode für einen Integrationstest

Service-Tests

Bei den Service-Tests wird getestet, ob der Service seine Aufgaben und somit auch seine Anforderungen erfüllt. Dafür werden Anfragen an den Service gestellt und geprüft, ob er diese verarbeiten kann und ein valides Ergebnis zurückliefert. Es wird auch überprüft, ob der Microservice auf Antworten von anderen Microservices sich wie vorgegeben verhält.

Damit diese Tests entwickelt werden können, müssen Anforderungen an den Microservice definiert werden. Besonders nützlich dabei ist, wenn eine Schnittstellenbeschreibung vorhanden ist. Bei den Microservices im Import-Bereich liegt bei allen eine Schnittstellenbeschreibung in Form einer Swagger-Beschreibung vor. Swagger ist ein Werkzeug, welches bei der Erstellung von solchen Beschreibungen unterstützt und eine eigene Beschreibung im YAML-Format anbietet. Im nachfolgenden Listing 5.3 ist die Swagger-Beschreibung für den Microservice “Tablebased-Importer" abgebildet.

```
swagger: '2.0'
info:
```

```
  description: Service to process uploaded table based imports
  version: '2.0'
  title: Table based importer
host: localhost:5555
basePath: "/"
tags:
- name: tablebased-import-controller
  description: Tablebased Import Controller
paths:
  "tablebased/{dataId}":
    delete:
      tags:
      - tablebased-import-controller
      summary: Deletes all persisted data for the given data id.
      operationId: deleteTablebasedFIleUsingDELETE
      consumes:
      - application/json
      produces:
      - "*/*"
      parameters:
      - name: dataId
        in: path
        description: dataId
        required: true
        type: string
      responses:
        '200':
          description: OK
        '204':
          description: No Content
        '401':
          description: Unauthorized
        '403':
          description: Forbidden
```

Listing 5.3: Beispiel Kontrakt als Swagger-Datei

Es existieren mehrere Programme, die das Testen von Schnittstellen anbieten wie z.B. Postman und SoapUI. Postman wurde bereits im Hauptprojekt eingesetzt, wobei es einen positiven Eindruck hinterließ [Pie17a]. Es sind keine Gründe bekannt, warum nicht alle Microservices mit Postman getestet werden können.

Postman kann aus den Swagger-Dateien automatisch Testfälle erstellen. Solange keine Anforderungen an den Microservice definiert sind, ergibt es Sinn, die Tests manuell auszuführen. Dabei bietet sich auch hier Exploratory Testing an, denn mit den gesammelten Informationen können die Anforderungen definiert und damit auch automatisierte Tests erstellt werden. Dies ist möglich, weil die Beschreibung der Schnittstelle nur Felder und ihre Typen wiedergibt, aber keine Werte. Mit Exploratory Testing können Werte für die Tests bestimmt werden.

Diese Tests können unter bestimmten Voraussetzungen lange dauern. Es wird jedoch keine Grenze bei der Dauer vorgegeben. Daher muss sich das Testen auf wichtige Kernfunktionen beschränken. Mit der manuellen Ausführung kann erarbeitet werden, welches die Kernfunktionen sind. Ergänzend dazu können auch die Aufrufe gefiltert werden, die länger dauern als die anderen Aufrufe.

Neben dem manuellen Ausführen bietet Postman auch das automatisierte Ausführen von Tests aus. Postman muss dabei so konfiguriert werden, dass die manuellen Tests automatisiert ausgeführt werden. Mit entsprechenden Kommandozeilen-Befehlen kann Postman in die Deployment Pipeline integriert werden.

In den beiden Abbildungen 5.2 und 5.3 ist ein Test abgebildet, der beim Metadata-Service das Erstellen eines Metadata-Objektes testet. In der ersten Abbildung werden die Werte für den POST-Aufruf gefüllt und in der zweiten Abbildung werden die Asserts bestimmt. Die Ausführung der Asserts bestimmt, ob ein Test erfolgreich ist oder nicht. In diesem Beispiel sind sehr einfache Asserts abgebildet. Theoretisch ist es auch möglich, auf genaue Werte innerhalb eines zurückgegebenen Objektes zu testen.

Werden andere Services benötigt, müssen diese simuliert werden. Dafür muss zuerst analysiert werden, welche anderen Microservices der unter Test stehende Microservice aufruft. Dann muss für den Microservice eine Umgebung geschaffen werden, in der die benötigten Microservices als Mock oder Stubs vorhanden sind.

Neben den Service-Tests sollen hier auch Consumer Driven Contract Tests ausgeführt werden. Dazu bietet sich auch die Technologie PACT an, welche bereits im Hauptprojekt untersucht worden ist [Pie17a]. Für das Entwerfen dieser Tests sind die gewonnen Informationen aus den manuellen Postman Tests sehr hilfreich, weil sie als Grundlage für das Implementieren der Tests dienen.

Zuständig für die Service-Tests und CDCT ist das Team, das den Microservice entwickelt. Deren Aufgabe ist es auch Stubs oder Mocks von ihrem Microservice bereit zustellen.

Im ersten Schritt müssen die Service-Tests manuell ablaufen, damit mit den gewonnenen Informationen die automatisierten Tests erstellt werden können. Sobald diese automatisierten

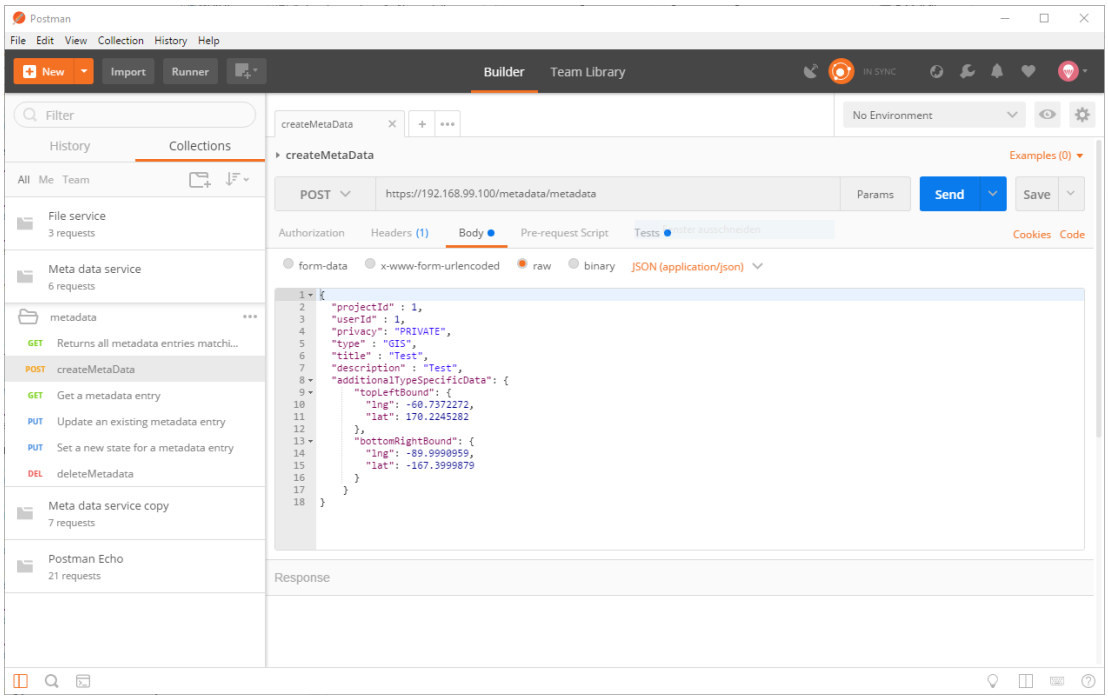

Abbildung 5.2: Erstellung eines Requestes zur Erzeugung von Metadata-Daten in Postman

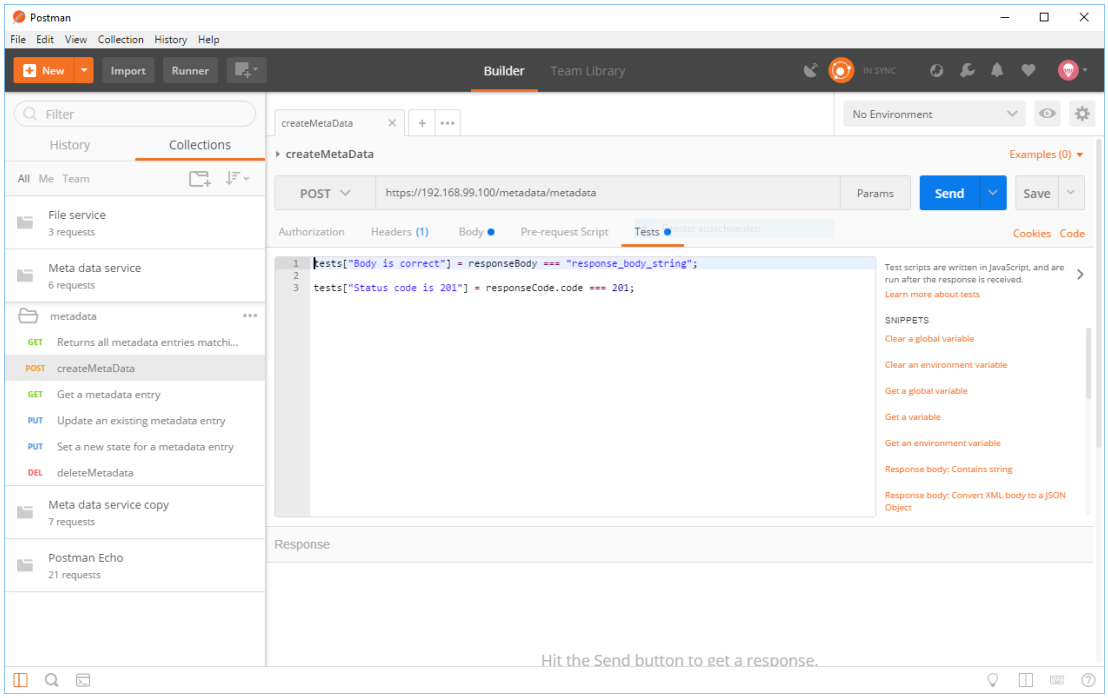

Abbildung 5.3: Asserts zum Testen der Ergebnisse des Requestes

Tests existieren, sollen ausschließlich nur noch diese ausgeführt werden. Besonders innerhalb der Deployment Pipeline werden nur automatisierte Service-Tests und CDCT stattfinden.

Bei neuen Funktionen in den Schnittstellen bietet sich an, diese zuerst wieder manuell zu testen und dann ggf. die Tests zu automatisieren. Dieses Verfahren findet jedoch außerhalb der Deployment Pipeline statt.

5.2.3 Testkonzept Gesamtsystem

Wie der Abbildung 5.4 entnommen werden kann, besteht das Testkonzept für das Gesamtsystem nur aus zwei Ebenen. Es gibt die Ebene Integrationstests und darauf aufbauend die Ebene der End-To-End Tests. Zusätzlich kann es manuelle Tests in Form von Exploratory Testing geben.

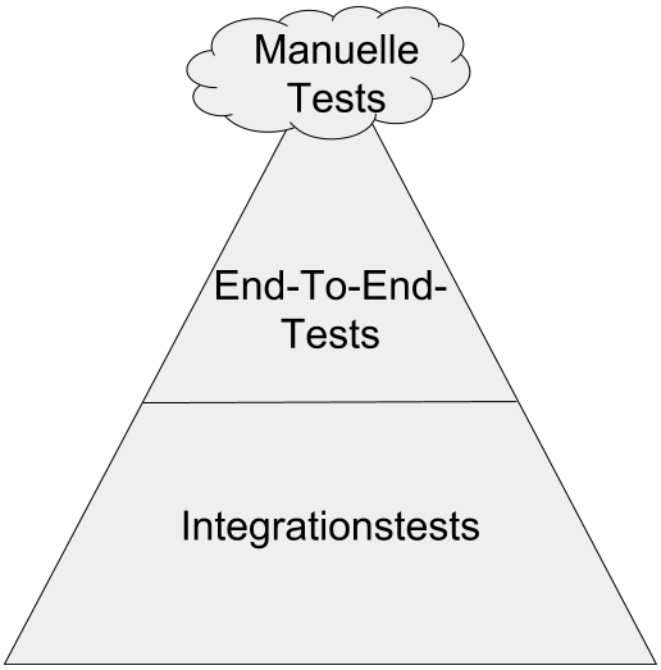

Abbildung 5.4: Testpyramide für das Gesamtsystem im MARS Framework

Integrationstest Gesamtsystem

Bei den Integrationstests wird nicht die Zusammenarbeit der einzelnen Services getestet. Hierbei soll ausschließlich geprüft werden, ob der Service korrekt veröffentlicht und gestartet wurde. Dazu gehören Prüfungen, ob der Services die anderen Microservices erreicht und im expliziten MARS-Umfeld sich beim Microservice Eureka angemeldet hat.

Dies ist ein von Netflix entwickelter Microservice, der die einzelnen Microservices verwaltet und auf Anfrage, die Adresse eines Microservices herausgibt. Als Beispiel fragt Microservice A nach der Adresse vom Microservice B beim Service Eureka an, um mit Microservice B kommunizieren zu können. Dafür muss Microservice B vorher dem Service Eureka seine Adresse mitgeteilt haben. Hat die Anmeldung von Microservice B bei Eureka nicht stattgefunden oder war fehlerhaft, dann kann Microservice A nicht mit Microservice B kommunizieren.

Um dies zu testen, kann eine Test-Anfrage an Eureka gestellt werden, z.B. nach den Adressen der anderen Microservices. Mit dieser Information können weitere Microservices angefragt werden, um zu prüfen, ob diese erreicht werden. In dem Zuge bietet es sich an, dass die Services Aufrufe für diese Tests bereitstellen, mit denen die Erreichbarkeit untereinander getestet werden kann. Zurzeit bieten keine Microservices im MARS Framework diese Methoden an, daher müssen die Services um diese Methode erweitert werden.

Diese Tests müssen automatisiert ablaufen, da die Dauer sehr gering sein soll. Ein Test auf dieser Ebene darf eine Laufzeit von einer Minute nicht überschreiten. Um dies zu gewährleisten kann wieder Postman verwendet werden. Mit Postman kann beim Eureka-Service angefragt werden, ob der Microservice sich angemeldet hat. Dazu kann über Postman, bei dem unter Test stehenden Microservice, die Methode aufgerufen werden, welche für Testzwecke prüft, ob der Microservice die anderen Services erreicht.

End-To-End Tests

Bei den End-To-End Tests wird die Gesamtfunktionalität über die Benutzeroberfläche getestet. Dabei soll sich das Testen auf wichtige Kernfunktionen beschränken. Diese werden dann via Regressionstests regelmäßig überprüft. Bei MARS könnten das das Anlegen eines neuen Projektes sowie der Import der notwendigen Dateien für die Simulation sein. Es werden nur funktionale Anforderungen getestet.

Für MARS reichen sehr wahrscheinlich höchstens fünf End-To-End Tests aus. Neben dieser Begrenzung der Anzahl der Tests ist es auch wichtig, eine Höchstdauer zu definieren. Generell sollte diese Ebene nicht länger als 20 min dauern, also höchstens vier Minuten pro Test bei fünf Tests.

Die End-To-End Tests laufen automatisiert ab. Trotz der berechtigen Kritik an Selenium, bietet es sich an, Selenium für die Implementierung dieser Tests einzusetzen.

Dazu werden generell keine Module simuliert. Nur in Ausnahmefällen ist dies erlaubt. Ausnahmen bilden Microservices, die entweder die Dauer der Tests zu sehr verlängern würden oder für ein instabiles Verhalten der Tests verantwortlich sind.

Im nachfolgenden Listing 5.4 ist ein Beispiel für ein Selenium End-To-End-Test, welcher mit Java und mit JUnit geschrieben ist.

Dies ist die Implementierung des Testfalls aus dem zweiten Szenario, bei dem die Untersuchung von Exploratory Testing Thema ist. Der Testfall ist in der Tabelle 4.15 beschrieben.

```
@Test
public void loadHelpPageTest() {
    driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);

    String expectedTitle = "Welcome!";

    String actualTitle = driver.findElement(By.
    cssSelector(".jumbotron h1")).getText();

    collector.checkThat(actualTitle,
    org.hamcrest.Matchers.equalTo(expectedTitle));
    meistern
    driver.findElement(By.cssSelector
    ("li.dropdown:nth-child(3) > a:nth-child(1)")).click();

    driver.findElement(By.cssSelector
    ("li.dropdown:nth-child(3) > ul:nth-child(2) >
    li:nth-child(1) > a:nth-child(1)")).click();

    expectedTitle = "Help";
    actualTitle = driver.findElement(By.tagName("h1")).getText();

    collector.checkThat(actualTitle,
    org.hamcrest.Matchers.equalTo(expectedTitle));

    String selectorTable = ".list-group";

    boolean isTable = !driver.findElements(By.
    cssSelector(selectorTable)).isEmpty();

    collector.checkThat(isTable,
    org.hamcrest.Matchers.equalTo(true));
}
```

Listing 5.4: Beispiel Selenium-Test

5.2.4 Deployment

Auf der Grundlage, dass jeder Microservice sein eigenes Testkonzept besitzt, verfügt jeder Microservice auch über seine eigene Deployment Pipeline. In der Deployment Pipeline sollen die einzelnen Testebenen durchlaufen werden. Die Deployment Pipeline ist in Abbildung 5.5 abgebildet. Sie endet mit der Ebene Service-Tests. Danach geht der Microservice in die Phase über, in der das Gesamtsystem getestet werden soll.

Abbildung 5.5: Deployment Pipeline für einen Microservice

Dieser Übergang ist in Abbildung 5.6 dargestellt. Es gibt für alle Microservices nur eine Teststufe Integrationstest und End-To-End Test auf dieser Ebene. Daher kann auch nur ein Service pro Zeit in diese Ebene übergehen. Andere Services müssen ggf. warten. Die beiden Umgebungen in den Tests sollen die Versionen von Microservices bereit halten, die sich aktuell in der Produktion befinden.

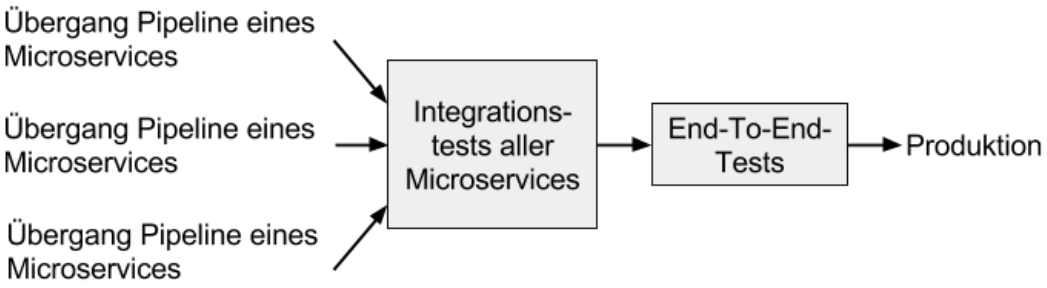

Abbildung 5.6: Deployment Pipeline für das Gesamtsystem im MARS Framework

Nach dem Abschluss der Deployment Pipeline werden manuelle Tests mit Exploratory Testing ausgeführt. Dabei sollen sie ausschließlich als Ergänzung zu den bisherigen Tests dienen. Sie sollen besonders dann eingesetzt werden, wenn menschliche Intuition gefordert ist. Zudem sollen sie für das Testen neuer Funktionen eingesetzt werden. Darüber hinaus sollen sie auch die Grundlage für die Entwicklung von End-To-End-Tests bilden.

5.3 Erläuterungen zum Testkonzept für das MARS Framework

In diesem Abschnitt wird begründet, warum das Testkonzept so entworfen worden ist. Zunächst wird erläutert, warum einzelne Ebenen existieren und warum sie so definiert sind. Dabei wird die gleiche Reihenfolge eingehalten wie bei der Vorstellung des Testkonzeptes. Im Anschluss daran wird beantwortet, wie das Testkonzept mit den verschiedenen Arten von Änderungen umgeht. Zuvor sollen jedoch zum besseren Verständnis noch allgemeingültige Aspekte begründet werden

Außer den Verweis auf CDCT gibt es keine expliziten Empfehlungen für Technologien, die eingesetzt werden sollen. Besonders auf den unteren Ebenen ergeben Vorschläge für Technologien wie z.B. JUnit keinen Sinn, weil zur Freiheit der Wahl der Programmiersprache bei der Entwicklung eines Microservices zählt auch die Freiheit der Auswahl der Technologie. Dabei enthält Technologie auch die Werkzeuge für das Testen. Daher müssen die Werkzeuge für das Testen von den Verantwortlichen selber ausgesucht werden. Auch wegen der möglichen Bandbreite von Programmiersprachen und Technologien können keine sinnvolle Vorschläge gemacht werden. Die in den Beispielen verwendeten Technologien dienen nur der Verdeutlichung und sollen bei der Auswahl der Technologie zur Testentwicklung Unterstützung bieten.

Die Ausnahme bei CDCT begründet sich damit, dass jeder Microservice einen Kontrakt definieren muss. Dieser Kontrakt ist grundsätzlich bei allen Microservices gleich aufgebaut. Daher kann in diesem Fall auch eine Technologie für das Testen empfohlen werden. Dasselbe gilt auch für Postman.

5.3.1 Unterteilung in zwei Ebenen & Deployment

Das Testkonzept für MARS ist in zwei unterschiedliche Bereiche unterteilt, wobei diese Entscheidung aus mehreren Gründen getroffen worden ist. Diese Entscheidung hat auch Auswirkungen auf den Deployment-Prozess.

Wie bereits mehrfach erwähnt, ist ein Microservice ein eigenständiges Programm, wobei erst die Zusammenarbeit der einzelnen Microservices die gesamte Anwendung ergibt. Aufgrund der Eigenständigkeit der Services muss jeder Service wie ein eigenständiges Programm gebaut, getestet und veröffentlicht werden.

Neben der Eigenständigkeit unterscheidet sich jeder Service in seinem Aufgabenbereich und dazu ggf. in der gewählten Programmiersprache und Technologie. Dadurch entstehen für jeden Service verschiedene Anforderungen an den Deployment-Prozess, welcher das Testen des Services enthält.

Genau um dies zu verdeutlichen, wurde das Testkonzept in zwei Ebenen unterteilt. Damit folgt das Testkonzept dem Ansatz von Wolff und genau genommen auch dem von Newman [Wol15, New16]. Bei Wolff wird eher eine parallele Struktur vorgesehen, während hier eher eine ergänzende Struktur verwendet wird.

Diese Unterteilung bietet den Vorteil, dass für jeden Service ein eigener Deployment-Prozess entworfen werden kann. Dies kann den Aufwand erhöhen, jedoch überwiegen die Vorteile. Zu einem besteht die Chance für jeden Service einen optimalen Prozess zur Veröffentlichung zu entwerfen und zum anderen können die Zuständigen für den Service dies grundsätzlich selber entscheiden. Vereinfacht gesagt, aus der Freiheit die Technologie für den Microservice frei zu wählen, generiert sich auch die Freiheit den Deployment-Prozess nach eigenem Belieben zu gestalten.

Theoretisch ist es möglich, denselben Deployment-Prozess für alle Services zu verwenden. Dies ist aber nicht empfehlenswert, weil je größer die Unterschiede der Services in Programmiersprache und Technologie sind, umso schwieriger wird es einen gemeinsamen Deployment-Prozess zu finden.

Dennoch müssen die Services zusammenarbeiten. Daher ist es wichtig, dass auch die Zusammenarbeit der Services im Deployment-Prozess dargestellt wird. Aus diesem Grunde existiert die zweite Ebene, die genau diese Aufgabe übernimmt.

Mit der Entscheidung für alle Microservices eine abschließende Ebene zu erstellen, bei der die Integrationstests für das Gesamtsystem und die End-To-End Tests ablaufen, werden unnötige Duplikate vermieden.

Existiert diese Ebene nicht und besitzt jeder Microservice seine eigenen Integrationstests für das Gesamtsystem und End-To-End-Tests, müssen für diese Tests entsprechende Testumgebungen geschaffen werden. Dadurch erhöht sich der Aufwand und es entstehen Duplikate, da immer die gleiche Testumgebung geschaffen werden muss. Mit diesem Weg wird der Argumentation von Newman gefolgt [New16].

Dazu wurde die Entscheidung getroffen, dass immer nur eine neue Version eines Services diese Ebene betreten darf. Sollte ein Fehler in dieser Ebene auftreten, kann die Fehlerquelle leichter gefunden werden, denn es wurde nur ein Service geändert.

Newman merkt dazu korrekt an, dass diese Strategie zu einem Flaschenhals führen könnte [New16]. Damit dies vermieden wird, wurden für den Integrationstest und die End-To-End Tests maximale Laufzeiten definiert.

5.3.2 Unit-Tests

Die Unit-Tests bilden die Grundlage für jede Art von Änderung. Diese Aufgabe übernehmen sie aus verschiedenen Gründen. Zu einem werden mit Unit-Tests klare Anforderungen an die zu testende Einheit definiert und zum anderen wird damit abgesichert, dass die Änderung nicht zu ungewollten Nebeneffekten führt.

Um Gewissheit zu haben, dass jede Funktion und Methode abgedeckt wird, wird Test-Driven-Development angewendet. Das ist der einfachste Weg, um dies sicherzustellen.

Dazu wurde die Annahme getroffen, dass die Unit-Tests die kürzeste Laufzeit von allen Testarten haben. Wie bei den Herausforderungen definiert soll das Testkonzept möglichst schnell Rückmeldung geben, ob die Tests bestanden sind. Daher bilden die Unit-Tests den größten Teil der Tests ab, um eine geringe Laufzeit der Tests zu gewährleisten.

Die Unit-Tests können jedoch nur geringe Laufzeiten ausweisen, wenn sie vollständig automatisiert ablaufen. Daher wurde entschieden, dass alle Unit-Tests automatisiert ablaufen sollen. Damit wird die gleiche Entscheidung vorgenommen, wie Clemson, Newman und Wolff sie getroffen haben [Cle14, New16, Wol15].

Mit dieser Entscheidung wird aber auch Whittaker widersprochen, der die Notwendigkeit sieht, manche Unit-Tests manuell ablaufen zulassen [WAC12]. Diese Notwendigkeit wird beim MARS Framework nicht gesehen. Denn zu einem wurde keine Methode oder Funktion gefunden, die beim Testen menschliche Intuition erfordert und zum anderen kann die maximale Laufzeit von einer Sekunde pro Test beim manuellen Unit-Tests nicht gehalten werden.

Darüber hinaus wurde das von Whittaker vorgestellte Konzept von definierten maximalen Laufzeiten für dieses Testkonzept übernommen [WAC12]. Dies soll auf der einen Seite auch sicherstellen, dass das Testen möglichst schnell abläuft. Auf der anderen Seite soll es dafür sorgen, dass die Tests isoliert ablaufen und schlank gehalten werden. Die Unit-Tests sollen nur eine Funktion bzw. Methode testen. Beim Brechen dieser Regel können die Tests länger als eine Sekunde dauern. Daher ist es auch eine Sicherheitsregel, damit die Definition der Unit-Tests nicht umgangen wird.

Die Wahl der zeitlichen Grenzen, wie in diesem Fall von einer Sekunde, orientierte sich an dem Testkonzept von Google [WAC12]. Die Zeiten bilden dabei einen ersten Orientierungspunkt, wie solch eine Grenze definiert sein kann. In der praktischen Anwendung muss jedoch geprüft werden, ob die zeitlichen Grenzen in ihrer aktuellen Festlegung Sinn ergeben. Daher müssen sie ggf. angepasst werden.

Die Isolierung der Tests ist wichtig und bildet die Basis für die geringe Laufzeit. Im Widerspruch zu Clemson wird die Notwendigkeit gesehen, alle zu testende Funktionen und Methoden zu isolieren. In manchen Szenarien kann die Isolierung einer Methode durchaus Aufwand

bedeuteten, aber die geringere Laufzeit und vor allem die Sicherheit, dass ausschließlich die zu testende Funktion überprüft wird und nicht deren Abhängigkeit, ist es den Aufwand wert. Dazu bilden besonders die Abhängigkeiten eine Herausforderung, die mithilfe der Isolierung gelöst werden soll.

Im Gegensatz zu den Testkonzepten von Newman, Clemson und Wolff wird keine Empfehlung ausgegeben, ob Mocks oder Stubs verwendet werden sollen. Die einzelnen Microservices unterscheiden sich zu sehr, um eine allgemeingültige Aussage zu treffen. Darüber hinaus soll diese Entscheidung von den Personen getroffen werden, die die Tests entwickeln.

Zusätzlich wurde entschieden, redundante Tests zu vermeiden. Diese Entscheidung wurde durch die wiederholte Verwendung des Metadata-Clients notwendig. Dieses Modul wird von mehreren Microservices verwendet. Jedoch sollen die Unit-Tests für die Methoden und Funktionen nur einmal definiert werden. Dazu muss klar festgelegt werden, wer für die Unit-Tests zuständig ist. Die Tests sollen jedoch immer ausgeführt werden, wenn ein Microservice, der dieses Modul verwendet, neu erstellt wird. Anderseits können unbeabsichtigte Nebeneffekte unentdeckt bleiben.

Wie gerade schon erwähnt, sind für die Unit-Tests klare Zuständigkeiten zu definieren. Der Entwickler bzw. die Entwickler einer Funktion/Methode müssen auch die dazugehörigen Unit-Tests schreiben. Bei nicht definierten Zuständigkeiten kann es passieren, dass keine Unit-Tests implementiert werden, weil sich niemand dafür zuständig fühlt.

Die von Clemson vorgestellte Unterteilung der Unit-Tests findet hier keine Anwendung. Clemson geht davon aus, dass jeder Microservice grundsätzlich die gleiche interne Architektur enthält. Dies Annahme kann bei dem MARS Framework nicht getroffen werden. Zu einem fehlt eine Dokumentation bzw. Guidelines zur Entwicklung eines Microservices, die sowas belegen könnten, und zum anderen konnte eine ähnliche interne Architektur in den näher untersuchen Microservices nicht festgestellt werden.

Vor allem wurde entschieden, alle Tests isoliert ablaufen zu lassen. Da die Isolierung der Unit-Tests als wichtige Basis angesehen wird, kann die Unterteilung von Clemson hier keine Anwendung finden.

5.3.3 Integrationtest Microservice

Die Existenz von Integrationstests innerhalb eines Microservices ist sehr nahe bei der Definition von Whittaker und Clemson, die ähnliches vorschlagen. Besonders die in dem Testkonzept für das MARS Framework verwendete Definition von Integrationstests innerhalb eines Microservices ist stark von der Definition von Googles Medium Tests beeinflusst. Ergänzt wird diese Festschreibung von der Definition von Clemson [WAC12, Cle14].

Beide Definitionen decken nur einen bestimmten Bereich ab. Bei Google wird die Zusammenarbeit von Methoden getestet und bei Clemson werden die Module getestet, die mit externen Ressourcen, wie anderen Microservices und Datenbanken, kommunizieren. Beide Aspekte sollen Beachtung finden und die Definitionen wurden deshalb zusammengeführt [WAC12, Cle14].

In den Testkonzepten von Newman und Wolff existieren Integrationstests gar nicht oder werden anders definiert [New16, Wol15]. Die Existenz von Integrationstests ist durch den Grundsatz begründet, dass möglichst auf den unteren Ebenen getestet werden soll. Das heißt, durch die Unit-Tests werden die einzelnen Funktionen und Methoden getestet, aber nicht deren Zusammenarbeit. Dafür existieren die Integrationstest.

Das Zusammenwirken der einzelnen Funktionen/Methoden wird auch durch Service-Tests oder End-To-End Tests mit abgedeckt. Dies ist jedoch nicht so performant und vor allem ist es bei Integrationstests innerhalb eines Microservices leichter, die Fehlerquelle zu finden, als auf den höheren Ebenen, da dort mehr Komponenten mit getestet werden.

Durch das Vorhandensein von Integrationstests müssen weniger Service-Tests und End-To-End Tests eingeplant werden.

Neben der Überprüfung der Zusammenarbeit von Methoden sollen besonders die Module getestet werden, welche die Kommunikation mit externen Ressourcen übernehmen. Besonders bei der Architektur Microservice ist die Kommunikation über das Netzwerk extrem wichtig. Ohne die Kommunikation über das Netzwerk können die Microservices nicht zusammenarbeiten, deswegen müssen diese Module ausführlich getestet werden.

Auch die Verarbeitung von Daten ist wichtig und sollte daher getestet werden. Die Einteilung von Clemson in “Persistence Integration-Tests" und in “Gateway Integration-Test" ist dafür sehr hilfreich [Cle14]. Diese Unterscheidung gibt dabei Unterstützung, um die Module ihren Aufgaben nach spezifisch prüfen zu können.

Diese Spezifizierung kann im Gegensatz zu Clemsons Einteilung bei den Unti-Tests angewendet werden, da davon ausgegangen werden kann, dass alle Microservices ein Modul für die Kommunikation besitzen. Ohne dieses Modul kann ein Microservice nicht kommunizieren. Hierdurch wäre er auch von der Gesamtanwendung isoliert und dementsprechend auch sinnlos.

Allerdings besitzen nicht alle Microservices eine Datenbank. Daher können bei diesen Microservices diese Tests wegfallen. Bei den untersuchten Microservices im MARS Framework besitzen fast alle Microservices eine Datenbank, deshalb sind die ‘Persistence Integration-Test" hier sinnvoll.

Wie bei den Unit-Tests laufen die Integrationstests komplett automatisiert ab. Dabei gilt eine ähnliche Begründung wie bei den Unit-Tests. Nur über die Automatisierung können die Tests effizient ablaufen, deshalb wird auch hier keine Notwendigkeit für manuelle Tests gesehen.

Für viel wichtiger ist, dass die Tests stabil ablaufen. Das Fehlschlagen von Tests darf nicht zur Gewohnheit werden, da ansonsten Fehler unentdeckt bleiben können und die Qualität der Anwendung gemindert wird. Damit einhergehend müssen die Integrationstests auch gewartet werden, damit sie stabil bleiben. Dies kann großen Aufwand bedeuten. Aus diesem Grund ist es besser, sich auf wenige stabile Tests zu konzentrieren, anstatt viele Tests einzusetzen, von denen einige möglicherweise instabil sind.

Für die Integrationstests müssen auch feste Zuständigkeiten definiert werden. Dies hat den großen Vorteil, dass es feste Zuständigkeiten für die Wartung gibt, damit festgelegte Personen oder Teams, diese Wartungen auch ausführen. Feste Zuständigkeiten können durchaus die Basis dafür bilden, dass die Integrationstests stabil ablaufen und aussagekräftige Ergebnisse liefern.

Grundlage für die Automatisierung der Tests ist die Isolierung der zu testenden Komponenten, andernfalls ist ein effizientes Testen nicht möglich. Auch hier gilt die gleiche Begründung wie bei den Unit-Tests. Auch in diesem Fall gibt es keine Empfehlung für Stubs oder Mocks, aus denselben Gründen wie bei den Unit-Tests.

Mit der Isolierung der Tests ist es auch möglich, dass die Integrationstests unter zehn Sekunden ablaufen. Wie bereits mehrfach erwähnt soll das Testen der Anwendung schnelles Feedback geben, dafür soll zu einem die zeitliche Grenze sorgen. Zum anderen soll das Zeitlimit ausufernde Integrationstests, die einen sehr großen Testumfang besitzen und aus diesem Grund auch viel Zeit in Anspruch nehmen, verhindern.

5.3.4 Service-Tests

Mithilfe der Unit-Tests und Integrationstests ist der Microservice ausführlich getestet. Aufgrund der Architektur und der damit verbundenen Grundlage, dass jeder Microservice ein eigenständiges Programm ist, ergibt sich die Notwendigkeit von Service-Tests.

In der Zusammenarbeit verlassen sich die anderen Microservices darauf, dass der Microservice seine Anforderungen erfüllt. Daher müssen diese auch explizit getestet werden, denn dadurch wird auch das Vertrauen in den Microservice erhöht. Die Service-Tests haben damit eine ähnlich Rolle inne wie die End-To-End Tests auf der Gesamtebene.

Je nach Aufgabe und Aufbau können Microservices unterschiedlich komplex sein. Aus diesem Grunde wird keine Grenze für die Laufzeit benannt, da sonst die Gefahr besteht, wichtige Service-Tests wegen dieser Vorgabe auszuschließen.

Bei den Service-Tests werden nur die anderen Microservices simuliert, jedoch keine internen sowie externen Komponenten des zu testenden Microservices. Aus diesen Gründen sollen auch nur Kernfunktionen des Services getestet werden, damit diese Ebene nicht zu lange dauert.

Um ausschließlich Kernfunktionen zu testen, müssen diese zunächst definiert werden. Diese Aufgabe muss das Team übernehmen, das den Microservice entwickelt. Das Team kann am besten einschätzen, welches die Kernfunktionen sind.

Dadurch ergibt sich von alleine, dass das Team für das Testen des Microservice zuständig ist. Dazu gesellen sich selbstverständlich weitere Aufgaben, wie das Entwerfen, Implementieren und Warten der Tests.

Des Weiteren soll das Team auch Mocks bzw. Stubs für die anderen Teams bereitstellen. Dies beruht auf der Annahme, dass das Team am geeignetsten für diese Aufgaben ist, da es den Microservice und seine Anforderungen am besten kennt.

Aus Gründen der geringeren Laufzeit werden auch hier hauptsächlich automatisierte Tests eingesetzt.

Beim Testen eines Microservices können Situationen entstehen, bei denen menschliche Intuition gefordert ist, wie z.B. bei sehr komplexen Anfragen. In diesen Fällen bieten sich manuelle Tests an. Diese können dafür verwendet werden, neue automatisierte Service-Tests zu entwerfen. Außerdem können sie auch für das Testen neuer Anforderungen genutzt werden.

Ohne eine Isolierung der anderen Microservices würden die anderen Microservices mit getestet werden. Dann wäre dies jedoch kein Service-Test mehr, sondern eher ein Integrationstest. Zudem verspricht die Isolierung des zu testenden Objekts einen Geschwindigkeitsvorteil, daher werden die anderen Microservices simuliert.

Neben dem Testen der Kernfunktionen muss auch unbedingt die Einhaltung der Kontrakte getestet werden. Sie bilden die Grundlage für die Kommunikation und damit auch die Zusammenarbeit mit den anderen Microservices. Bei einer Nichteinhaltung dieser Kontrakte können unerwartete Fehler auftreten und ggf. das komplette System betreffen. Mit dem Einsatz von Consumer Driven Contract Tests kann dies vermieden werden.

Das Einsetzen von CDCT wurde bereits im Hauptprojekt untersucht [Pie17a]. Als Ergebnis wurde festgehalten, dass CDCT einige Schwächen besitzt, aber effizient und ausführlich die Einhaltung der Kontrakte testet. Auf Grundlage dieser Erfahrungen wurde entschieden, in diesem Testkonzept CDCT zu verwenden, insbesondere da dadurch eine wichtige Herausforderung gemeistert wird, wie im Abschnitt zur Beantwortung der Herausforderungen erklärt wird.

5.3.5 Integrationstests Gesamtsystem

Das Konzept und die Definition von Integrationstests auf dieser Ebene sind inspiriert vom Testkonzept von Wolff, der ähnliche Integrationstests auf dieser Ebene definiert.

Im Gegensatz zu Newman und Clemson wird die Notwendigkeit von Integrationstests gesehen [New16, Cle14]. Jedoch nicht aus dem Grund, die Zusammenarbeit der Microservices zu testen. Dies wird durch die vorherigen Tests, besonders durch die Service-Tests abgedeckt. Das Testkonzept folgt der Annahme, dass sobald jeder Microservice seine Anforderungen erfüllt und sich wie erwartet verhält, dass dann auch die Zusammenarbeit funktioniert. Daher wird auf dieser Ebene nicht die Zusammenarbeit getestet, da als einzige neue Komponente das Netzwerk hinzu kommt. Dies führt zu langsameren Tests, ohne dass neue Erkenntnisse gewonnen werden. Daher soll auf dieser Ebene lediglich überprüft werden, ob die neue Version vom Microservice erfolgreich bereitgestellt und korrekt konfiguriert wurde.

Die Integrationstests bilden die Grundlage für die darauffolgenden End-To-End-Tests. Vermeintlich leichte Fehler, wie ein nicht erfolgreiches Deployment oder falsche Konfiguration, werden vor dem Starten der End-To-End-Tests ausgeschlossen, damit sich die End-To-End-Tests ausschließlich auf das Testen von funktionalen Anforderungen konzentrieren können.

Da das Testen der Bereitstellung und der Konfiguration nicht sehr aufwendig sind, dürfen Integrationstests nicht lange dauern. Daher auch die Definition von einer Höchstdauer von einer Minute.

Damit die Höchstdauer nicht überschritten wird, werden die Tests automatisiert ablaufen. Zudem wird das Kontrollieren der Bereitstellung der Konfiguration eine eher stupide Aufgabe sein, weshalb manuelles Testen hier keinen Sinn ergibt.

5.3.6 End-To-End Tests

Newman begründet in seinem Testkonzept sehr ausführlich, warum er End-To-End-Tests ablehnt und sie durch andere Tests ersetzen möchte. Die meisten seiner Argumente ergeben Sinn und können nachvollzogen werden. Trotzdem wird die Notwendigkeit gesehen, End-To-End-Tests auszuführen [New16].

Wie in der Testpyramide (Abbildung 4.4) von Newman dargestellt, erhöht sich das Vertrauen in die Anwendung mit dem Abschluss der Tests auf jeder weiteren Ebene nach oben. Da die End-To-End-Tests die Spitze dieser Pyramide abbilden, geben sie dementsprechend das größte Vertrauen in die Anwendung. Diese Aussage wird für dieses Testkonzept übernommen.

Auf jeden Fall sollte dieses Vertrauen nicht unterschätzt werden. Die End-To-End-Tests geben nicht nur den Entwicklern die Sicherheit, dass ihr Programm ordnungsgemäß funktioniert, sondern auch weiteren Personen, die in Verbindung mit dem Produkt stehen, wie z.B. Projektleitern. Besonders für technisch nicht vorgebildete Personen sind die Tests auf den unteren Ebenen schwer nachvollziehbar. Die End-To-End-Tests sind dagegen leichter zu

verstehen, da sie das Programm über die Oberfläche testen, so wie ein richtiger Anwender die Anwendung nutzen würde.

Aus den Erfahrungen, die im Grundprojekt bei der Untersuchung von End-To-End-Tests mit dem Framework Selenium und der hier ausgeführten Untersuchung zu Exploratory Testing gesammelt wurde, kann bestätigt werden, dass End-To-End-Tests instabil und langsam sein können. Dies ist jedoch kein Ausschlusskriterium, da der Vorteil des gewonnen Vertrauen überwiegt.

Daraus schlussfolgend wurde die Lösung gewählt, nur wenige, aber dafür stabile End-To-End-Tests zu entwickeln. Daher werden eine maximal Anzahl von Tests und ergänzend dazu auch eine Höchstdauer definiert. Dadurch wird sichergestellt, dass nur wenige Tests vorhanden sind und diese nicht zu lange dauern. In diese Ebene darf immer nur ein neuer Microservice zur Zeit eintreten. Um keinen zu großen Flaschenhals zu erzeugen, müssen diese Grenzen unbedingt eingehalten werden.

Die Entscheidung zur Begrenzung auf fünf Testfälle für das MARS Framework basiert auf den verschiedenen Funktionen, die in der Oberfläche bereit stehen. Es gilt die Annahme, dass mithilfe von fünf Testfällen die Kernfunktionen in der Oberfläche getestet werden können.

Die Kernfunktionen werden mithilfe von user journeys und ähnlichem definiert. User journeys enthalten alle Schritte, die ein Nutzer anwenden muss, um zu einem bestimmten Ziel zu gelangen. Sie beschreiben die Anwendung des Programmes durch den Nutzer. Im MARS Framework kann dies z.B. das Importieren von Modellen sein.

Aus diesem Grund werden auf dieser Ebene nur automatisierte End-To-End Tests verwendet, die für die Regressionstests gedacht sind. Beim Exploratory Testing dagegen soll sich der Tester so viel Zeit wie nötig nehmen, weshalb dies nicht Bestandteil der Deployment Pipeline sein kann, da die zeitliche Begrenzung wahrscheinlich nicht eingehalten werden kann.

Die Aufgabenverteilung zwischen automatisierten Tests und Exploratory Testing basiert auf den Untersuchungen, die in dieser Arbeit vorgenommen worden sind.

Das Simulieren von Microservices kann von Vorteil sein, wenn es externe Microservices sind. Dies sollte unbedingt eingeplant werden, wenn dies zu stabileren Tests führt.

5.3.7 Beantwortung der Herausforderungen

Mit dem in dieser Arbeit entwickelten Testkonzept sollen für die verschiedenen Herausforderungen, wie sie im Abschnitt 4.1 beschrieben worden sind, Lösungen gefunden werden. In diesem Abschnitt wird untersucht, ob es dem Testkonzept gelungen ist.

Die erste Herausforderungen ist, dass die Architektur Microservices sehr ähnlich wie ein verteiltes System ist. Diese Problematik wird auf verschiedenen Ebenen angegangen.

Besonders die Aufteilung des Testkonzeptes ist eine Antwort darauf. Jeder Microservice wird wie ein eigenständiges Programm getestet. Mithilfe der verschiedenen Testebenen bei einem Microservice wird sichergestellt, dass die einzelnen Microservices ihre Aufgaben erfüllen sowie mit anderen Microservices kommunizieren können.

Dadurch wird es nicht notwendig die Zusammenarbeit einzelner Microservices zu testen. Wenn Microservice A und B wie erwartet funktionieren und ihre Tests bestehen, dann ist davon auszugehen, dass auch die Zusammenarbeit keine Probleme bereitet.

Zwar kommt im produktiven Betrieb das Netzwerk hinzu, jedoch wird mit den Integrationstests bei einem Microservice auch geprüft, ob die Microservices mit typischen Netzwerkfehlern wie z.B. Timeouts zurecht kommen. Insbesondere mit Clemsons Gateway Integration-Test wird dies sichergestellt.

Zusätzlich werden mit dem Integrationstest für das Gesamtsystem kontrolliert, dass die neue Version eines Microservices korrekt veröffentlicht wurde und die anderen Microservices erreicht werden. Auch das ist eine Beantwortung der Herausforderung des verteilten Systems.

Mit nur einer Testumgebung für alle Microservices für die Integrationstests für das Gesamtsystem und für die End-To-End Tests wird der Aufwand für diese Tests minimiert. Es muss nicht für jede neue Version eines Microservices eine neue Umgebung aufgebaut werden. Dies kann sonst bei einer Microservice-basierten Anwendung sehr aufwendig sein.

Bei den End-To-End-Tests kann das verteilte System ein Problem darstellen. Aufgrund der Kommunikation über das Netzwerk können die Tests deutlicher langsamer ablaufen, daher wurde eine Obergrenze an Tests und an der Gesamtdauer definiert.

Mit der Aufteilung des Testkonzeptes wird auch der Besonderheit der starken Modularisierung in der Architektur Microservices Beachtung geschenkt. Dies passiert insbesondere durch den Ansatz, dass jeder Microservice eine eigene unabhängige Deployment Pipeline besitzt.

Des Weiteren wurde die Herausforderung definiert, dass das Testkonzept ein schnelles Testen der Anwendung ermöglichen soll. Um diese Herausforderung zu meistern, wurden für die Unit-Tests, die Integrationstests und für die End-To-End-Tests feste Zeitgrenzen definiert.

Weiterhin wurde festgelegt, dass die meisten Tests vom Typ Unit-Tests sein müssen. Die Unit-Test besitzen die geringste Dauer von allen Tests. Mit einem starken Fokus auf Unit-Tests wird das Meistern dieser Herausforderung sehr gut unterstützt.

Ebenso sind alle Tests als automatisierte Tests vorgesehen. Die einzige Ausnahme bilden die manuellen End-To-End-Tests. Diese sind jedoch außerhalb der Deployment Pipeline und verzögern dadurch nicht den Ablauf.

Außerdem werden bei den Unit-Tests, Integrationstests innerhalb eines Microservices und bei den Service-Tests die Abhängigkeiten simuliert. Mit diesen Schritten ist ein effizientes Testen möglich.

Die Simulation von Abhängigkeiten ist selber als Herausforderung genannt. Die Lösung dafür ist, dass der Entwickler bzw. die Teams von ihrem Microservice eine Testkopie für die anderen Teams bereitstellen. Sie kennen ihren Microservice am besten und sind dafür am geeignetsten, eine Kopie davon zu erstellen, die sich wie das Original verhält.

Zusätzlich müssen die anderen Teams nicht von jedem Microservice, den sie verwenden eigene Kopien erstellen. Dadurch werden auch unnötige Duplikate zu vermieden.

Des Weiteren ist als Herausforderung benannt, das nötige Wissen über die vorhandenen Abhängigkeiten zu sammeln. Diese Herausforderung ist sehr eng verknüpft mit der Herausforderung, dass beim MARS Framework die Dokumentation veraltet oder nicht vorhanden ist. Für diese Herausforderung konnte keine umfassende Lösung gefunden werden.

Grundsätzlich ist es nicht die Aufgabe eines Testkonzeptes für eine ausreichende Dokumentation zu sorgen. Trotzdem fällt besonders beim Testen das Fehlen einer solchen Dokumentation auf. Das Testen kann dabei unterstützen, Schwerpunkte festzulegen, an denen eine Dokumentation erstellt werden muss.

Ergänzend kann Exploratory Testing dazu beitragen, dieses Wissen aufzubauen. Dennoch kann damit eine nicht vorhandene Dokumentation nicht ersetzt werden.

Eine weitere Herausforderung bei der Simulation von anderen Microservices ist, wenn keine lose Kopplung zwischen den Microservices vorherrscht. Auch dieses Problem ist nicht durch ein Testkonzept zu lösen. Allerdings fallen aufgrund der strengen Vorgabe, andere Module bzw. Services zu simulieren, solche Zustände früher auf und können dann ggf. leichter behoben werden.

Da für die einzelnen Microservices unterschiedliche Teams zuständig sind, existiert eine weitere Herausforderung. Diese wird unter anderem mit der Festlegung klarer Verantwortungen auch für die Testentwicklung und alle weiteren Aufgaben in Zusammenhang mit den zu testenden Microservices gelöst.

Wie bereits erwähnt, sind die Teams dazu verpflichtet, für die anderen Teams Testkopien von ihrem Microservice bereit zu stellen. Aufgrund dieser Entscheidungen müssen sich die Teams erst bei den Integrationstests auf der Gesamtebene und bei den End-To-End-Tests absprechen.

Auch das Testen der Schnittstelle ist als Herausforderung definiert. Als Lösung bieten sich Consumer Driven Contract-Tests an. Diese wurden bereits im Hauptprojekt untersucht. Dieser Testansatz eignet sich sehr gut dazu, Schnittstellen effizient und ausführlich zu testen [Pie17a].

Wie bei den spezifischen Herausforderungen bezüglich des MARS Frameworks vorgestellt, erhält das MARS Framework ständig Änderungen und dabei teilweise sehr große, die die gesamte Infrastruktur betreffen. Für diese Herausforderung existieren mehrere verschiedene Lösungen. Das Testkonzept ist relativ unabhängig von der Infrastruktur. Die Unit-Tests und Integrationstests werden nicht durch die Infrastruktur beeinflusst. Beide sollen beim Erstellen des Microservices ausgeführt werden. Daher ist für sie nicht relevant, ob der Microservice via Docker oder kubernetes gestartet wird. Änderungen an der Infrastruktur, wie z.B. die Verwendung von kubernetes, hat auf die anderen Testebenen nur indirekten Einfluss.

Alle Testarten werden innerhalb der Deployment Pipeline ausgeführt. In dieser Deployment Pipeline wird der Microservice auch bereitgestellt. Eine Änderung an der Infrastruktur bedeutet somit eine Änderung der Bereitstellung und somit auch der Deployment Pipeline. Die anderen Testebenen werden alle nach der Veröffentlichung ausgeführt. Daher werden auch sie durch die Änderung an der Infrastruktur beeinflusst. Trotzdem sollte eine solche Veränderung keinen Einfluss auf den Ablauf der Tests besitzen.

Das Kapseln von Aufrufen im MARS Framework, wie am Beispiel des Objektes Metadata-Clients, bildet eine weitere Herausforderung. Da das Objekt keine komplexen Strukturen enthält, die unter bestimmten Umständen zu langläufigen Tests führen könnten, wurde entschieden, dass das Objekt beim jedem Service, der es verwendet, mit zu testen ist. Zum einen ist der zusätzliche Aufwand überschaubar und zum anderen wird sichergestellt, dass das Objekt in jedem Microservice, der es verwendet, funktioniert.

Das Testen des Objektes funktioniert aber nur, wenn feste Zuständigkeiten definiert sind. Da das Objekt bei verschiedenen Microservices verwendet wird, muss geklärt werden, wie die Zuständigkeiten aussehen.

Die letzte Herausforderung betrifft das Testen der Microservices, die nur bei der Simulation Verwendung finden. Für diese Problematik wurde keine Lösung gefunden. Denn wie bereits am Anfang der Vorstellung des Testkonzeptes beschrieben, ist diese Thematik zu umfangreich. Daher konnte das Thema nicht genauer untersucht werden.

5.3.8 Umgang mit Änderungen

Neben der Beantwortung der Fragen zu den Herausforderungen wurde auch die Anforderung an das Testkonzept gestellt, dass es mit den verschiedenen Arten von Änderung umgehen kann. In diesem Abschnitt wird vorgestellt, wie das Testkonzept mit den verschiedenen Arten von Änderungen umgeht.

Grundsätzlich gilt für alle Änderungen, dass immer die komplette Deployment Pipeline ausgeführt wird. Nur darüber kann sichergestellt werden, dass die Änderung zu keinem Fehler führt.

Aufgrund der Fokussierung des Testkonzeptes auf ein schnelles und effizientes Testen ist es möglich, die komplette Deployment Pipeline immer wieder auszuführen.

Hinzufügen eines neuen Features

Beim Hinzufügen eines neuen Features ist die Ebene entscheidend, auf der das Feature hinzugefügt wird.

Wird eine neue Methode hinzugefügt, dann muss für die Methode auch ein neuer Unit-Test entworfen und implementiert werden. Dazu muss geprüft werden, ob die Zusammenarbeit der neuen Methode mit den bestehenden Methoden, Bestandteil eines neuen Integrationstest sein muss.

Bei einem neuen Feature in der Schnittstelle müssen die Service-Tests und vor allem die Consumer Driven Contract Tests angepasst werden. Da der Microservice mit seiner neuen Methode in der Schnittstelle als Provider dient, müssen auch die Consumer Driven Contract Tests bei seinen Consumern angepasst werden.

Ist das neue Feature in der Oberfläche hinzugefügt worden, muss nach Abschluss der Deployment Pipeline das neue Feature mit Exploratory Testing untersucht werden. Dabei soll explizit geprüft werden, ob das neue Feature die Anforderung erfüllt. Gehört das neue Feature zu den Kernfunktionen der Anwendung, dann muss auf Grundlage von Exploratory Testing ein neuer End-To-End-Tests geschrieben werden.

Das neue Feature kann alle Ebenen betreffen. In diesem Fall müssen für alle Ebenen die eben beschriebenen Schritte ausgeführt werden.

Fehlerbehebung

Wie der Fehler festgestellt wurde, hat Einfluss darauf, wie darauf reagiert werden soll, denn das Feststellen eines Fehlers durch einen fehlgeschlagenen Test veranlasst, dass nach der vermeintlichen Fehlerbehebung die Deployment Pipeline erneut ausgeführt wird. Sind dabei alle Tests erfolgreich, wurde der Fehler erfolgreich behoben.

Anders sieht es aus, wenn der Test erst in Produktion oder beim Exploratory Testing aufgefallen ist. Liegt die Fehlerquelle auf Unit- der Integrations- oder Service-Ebene, muss ein entsprechender Test auf dieser Ebene hinzugefügt werden. Der Test muss vor dem Beheben des Fehlers hinzugefügt werden. Dadurch ist es möglich zu prüfen, ob der neue Test den Fehler abdeckt.

Ist der Ursprung des Fehlers auf der End-To-End-Ebene, muss entschieden werden, ob die Anforderung durch einen automatisierten Test abgedeckt werden soll. Dafür muss die fehlerhafte Funktion zu den Kernfunktionen gehören. Daher muss hier von Fall zu Fall entschieden werden.

Änderung einer Anforderung

Erhält eine Anforderung eine Änderung, dann werden die dazugehörigen Tests aktualisiert, so dass sie mit der Änderung konform sind.

Existieren bisher keine Tests, muss zuerst geprüft werden, warum keine Tests existieren. Die Unit-Tests bilden die Grundlage für jede Art von Änderung. Daher müssen diese hinzugefügt werden, sollten sie fehlen.

Bei den anderen Tests muss im Einzelfall entschieden werden, ob durch die Änderung Tests notwendig geworden sind.

Ersetzen eines Microservices

Eine weitere Art von Änderung ist, wenn ein Microservice durch einen anderen Microservice ersetzt wird. In diesem Fall müssen die Consumer Driven Contract-Tests bei den Consumern des alten Microservices angepasst werden. Ersetzt der neue Microservice den alten Microservice komplett, dann muss bei den Consumern der neue Microservice als Provider eingetragen werden.

Dies ist vor allem dann notwendig, wenn sich die Schnittstelle ändert. Eine Änderung der Schnittstelle hat auch Einfluss auf die Service-Tests der Consumer. Bei denen muss zum einen der Mock bzw. Stub ausgetauscht werden. Zusätzlich müssen die simulierten Anfragen an den Stub/Mock aktualisiert werden und ggf. auch die Antworten.

Es wird davon ausgegangen, dass der neue Microservice bereits Unit-Tests, Integrationstests und Service-Tests enthält. Eine Anpassung der End-To-End-Tests ist nicht notwendig, weil diese Tests davon unberührt sind. Durch eine Ersetzung eines Microservices verändern sich nicht die Anforderungen an das Gesamtsystem. Sollte dennoch ein neues Feature hinzugefügt werden oder eine Anforderung geändert werden, gilt das gleiche Verhalten, wie oben beschrieben.

Einführung eines neuen Microservices

Das Verhalten bei der Einführung eines neuen Microservices ist sehr ähnlich wie beim Ersetzen eines Microservices.

Für die Consumer des neuen Microservices werden die Consumer Driven Contract Tests angepasst. Dazu gilt auch hier, dass beim Hinzufügen eines neuen Features oder beim Ändern einer Anforderung das gleiche Verhalten gilt, wie oben beschrieben.

Das gleiche gilt auch für die Service-Tests. Diese müssen, wie oben beschrieben, angepasst werden. Auch hier wird davon ausgegangen, dass der neue Microservice bereits Unit-Tests, Integrationstests und Service-Tests besitzt.

Wegfall eines Microservices

Ähnlich wie bei den zuvor beschriebenen Arten von Änderungen auch, müssen beim Wegfall eines Microservices, die Consumer Driven Contract Tests bei den Consumern aktualisiert werden.

Daneben müssen auch hier wieder die Service-Tests angepasst werden.

Fallen durch den Wegfall des Microservices auch Funktionen weg, dann müssen ggf. End-To-End-Tests angepasst oder gelöscht werden.

5.3.9 Handhabung der Besonderheiten

Das entwickelte Testkonzept sollte zusätzlich die Anforderung erfüllen, die Besonderheiten der Architektur Microservices zu unterstützen und vor allem nicht zu behindern. Deswegen werden in diesem Abschnitt die einzelnen Besonderheiten untersucht, wie das Testkonzept diese Anforderungen handhabt.

Da die meisten Aspekte dazu bereits in den vorherigen Abschnitten genannt wurden, soll dieser Abschnitt dazu dienen, die verschiedenen Aspekte zusammenzufassen, damit der Umgang mit den Besonderheiten einmal kompakt im Zusammenhang dargestellt wird.

Eine große Besonderheit bzw. ein großer Vorteil ist die starke Modularisierung der Architektur Microservices. Dieser Vorteil wird durch die Aufteilung des Testkonzeptes unterstützt. Mit der Aufteilung ist es möglich, für jeden Microservices eine eigene spezifische Deployment Pipeline und Teststrategie zu entwerfen.

Darüber wird auch die starke Unabhängigkeit der Microservices unterstützt, weil die einzelnen Deployment Pipelines nicht voneinander beeinflusst werden. Mit Unabhängigkeit ist hier gemeint, dass die Microservices untereinander unabhängig sind und dass die Microservices unabhängig voneinander erstellt und veröffentlicht werden können.

Dennoch sind die Microservices in dem hier entwickelten Testkonzept nicht komplett unabhängig voneinander. Die Zusammenführung der verschiedenen Deployment Pipelines zu einer Deployment Pipeline, in der die Integrationstests für das Gesamtsystem und die End-To-End Tests ausgeführt werden, sind der Grund dafür. Dadurch entsteht eine Abhängigkeit

untereinander, da diese Ebene erst betreten werden darf, wenn sich kein anderer Microservice auf dieser Ebene befindet. Um dieses Problem gering zu halten, sind für beide Testebenen zeitliche Grenzen vorgegeben worden.

Mit der Aufteilung wird auch die Technologiefreiheit der einzelnen Microservices unterstützt. Für die Implementierung der Tests können jeweils die Technologien gewählt werden, die am geeignetsten dafür sind. Die anderen Microservices werden von dieser Entscheidung nicht betroffen.

Trotzdem müssen auf der Ebene der Tests für das Gesamtsystem gemeinsame Technologien verwendet werden. Auf dieser Ebene ergibt es keinen Sinn, verschiedene Technologien zu verwenden. Mit der Verwendung derselben Technologien auf dieser Ebene können z.B. zusätzliche Wartungsaufwände vermieden werden, die durch die Verwendung verschiedener Technologien entstehen können. Insbesondere die Technologie für das Stuben der Microservices ist ein gutes Beispiel dafür. Verwendet jedes Team eine andere Technologie dafür, muss sich jedes Team in die verschiedenen Technologien einarbeiten und ist dann auch für die jeweilige Wartung zuständig. Das ist mehrfacher Arbeitsaufwand, daher ergibt es Sinn, sich auf eine Technologie zu einigen.

Der betriebliche Abstimmungsbedarf wird dadurch nicht unbedingt größer. Selbstverständlich muss abgesprochen werden, wie die Integrationstests auf der Gesamtebene und die End-To-End Tests aussehen. Jedoch müssen die Teams in jedem Fall besprechen, wie ihre Microservices zusammenarbeiten, um die Funktionen für das Gesamtsystem bereitstellen zu können. Dabei kann auch geklärt werden, welche Technologie für das Stuben verwendet werden soll und wie die Tests für das Gesamtsystem aussehen sollen. Im Gegensatz dazu müssen beim Testen eines einzelnen Microservices keine Absprachen getroffen werden, da jedes Team unabhängig voneinander testen kann.

Auch die leichte Ersetzbarkeit der Microservices wird vom Testkonzept unterstützt. Wie im Abschnitt 5.3.8 vorgestellt, kann das Testkonzept mit dieser Art von Änderung sehr gut umgehen. Änderungen sind nur bei den Service-Tests und CDCT notwendig. Da aufgrund der Ersetzung bei den betroffen Microservices die Schnittstellen stets angepasst werden müssen, können diese Änderungen simultan ausgeführt werden.

Mit dem Fokus auf Unit-Tests und die zeitlichen Grenzen bei fast allen Tests wird der Vorteil des schnellen Erstellens und Veröffentlichens weiter unterstützt. Das gesamte Testkonzept ist darauf ausgelegt, effizient abzulaufen. Daher finden die meisten Tests auch in einer isolierten Umgebung statt.

Der Vorteil der Skalierung hat die Entwicklung des Testkonzeptes nicht beeinflusst. Dieser Vorteil existiert ausschließlich im produktiven Betrieb.

5.4 Ausblick

Dieser Abschnitt gibt Ausblicke darüber, wie das Testkonzept für bestimmte Anwendungsfälle angepasst werden müsste. Hierbei werden drei Anwendungsfälle untersucht. Beim ersten Anwendungsfall wird beschrieben, welche Anpassungen vorgenommen werden müssen, wenn eine aktuelle Version des MARS Frameworks getestet werden soll. Ähnlich ist dabei auch der zweite Anwendungsfall. Hier wird dargelegt, welche Modifizierungen vorgenommen werden müssen, wenn das Testkonzept auf alle Bereiche im MARS Framework angewendet werden soll. Im letzten Anwendungsfall wird ausgeführt, wie die Veränderungen aussehen müssen, wenn des Testkonzept für ein allgemeines Microservice-basiertes System benutzt wird.

5.4.1 Verwendung der aktuellsten Version vom MARS Framework

Wie am Anfang dieses Kapitels erwähnt wurde, ist das Testkonzept auf Basis einer Version des MARS Frameworks erstellt, welche aus dem Dezember 2016 stammt. In der Zwischenzeit gab es einige Veränderungen am MARS Framework, die nicht mehr mit eingeflossen sind. Dieser Ausschnitt gibt einen Ausblick darüber, wie das Testkonzept angepasst werden muss.

Die größte Umstellung beim MARS Framework betraf die Infrastruktur. Zwar wurde auch bereits in der Version aus Dezember 2016 kubernetes verwendet. In der jetzigen Version ist die Verwendung von kubernetes allerdings noch verbreiteter. Diese Umstellung betrifft die einzelnen Testdefinitionen nicht. Sie beeinflusst aber die Deployment Pipeline und die Integrationstests auf der Ebene des Gesamtsystems.

Anhand dieser Änderungen muss die Deployment Pipeline dahingehend angepasst werden, dass die Microservices korrekt im Cluster bereitgestellt werden. Daraus generiert sich eine weitere Anforderung an die Integrationstests. Diese müssen nun auch prüfen, ob der Microservice korrekt im Cluster veröffentlicht worden ist und die anderen Services im Cluster erreicht.

Des Weiteren wurden einzelne Services durch andere Microservices ersetzt oder aufgeteilt. Für diese Veränderungen müssen Anpassungen vorgenommen werden, wie sie im Abschnitt 5.3.8 Umgang mit Änderungen beschrieben worden sind.

5.4.2 Ausweitung auf alle Bereiche im MARS Framework

Bei einer Anwendung des Testkonzeptes auf alle Bereiche im MARS Framework müssen verschiedene Tätigkeiten erfolgen. Zuallererst muss ergründet werden, ob die Annahme, dass alle Bereiche sich ähnlich verhalten wie der Import-Bereich, korrekt ist. Dafür muss zunächst Wissen über die anderen Bereiche gesammelt werden.

Es ist wahrscheinlich, dass auch in diesen Bereichen keine aktuelle oder gar keine Dokumentation vorliegt. Daher wird der Einsatz von Eploxoratory Testing wieder notwendig sein. Ist das Ergebnis, dass die Annahme korrekt war, müssen keine gravierenden Änderungen vorgenommen werden.

Andernfalls muss das Testkonzept angepasst werden. Dabei ist es am wahrscheinlichsten, dass Änderungen an den Service-Tests und den Tests für das Gesamtsystem durchgeführt werden müssen. Mit diesen Tests werden die Funktionsweisen der einzelnen Microservices und des Gesamtsystems überprüft. Auf Grundlage der getroffenen Annahme wurde davon ausgegangen, dass diese Tests ausreichen, um die Zusammenarbeit der Microservices sicher zustellen. Wenn die Annahme nicht korrekt war, kann dies nicht sicher gesagt werden. Es werden dann zusätzliche Informationen benötigt.

Die Unit-Tests und Integrationstests sollten von der Korrektheit der Annahme nicht betroffen sein. Diese Tests befinden sich intern im Microservice. Es werden interne Methoden und das interne Zusammenarbeiten getestet. Daher sollten sie nicht durch äußere Zustände beeinflusst werden.

Mit einer Ausweitung des Testkonzeptes können die Informationen aus dem Import-Bereich Grundlagen für die anderen Bereiche bilden. Insbesondere das Erstellen der Deployment Pipelines und der verwendeten Technologien können als Vorlage dienen. Dabei gehört die Aufgabe dazu, für die weiteren Bereiche zu prüfen, welche Technologien in Frage kommen. Während im Import-Bereich die Microservices mit Java geschrieben worden sind, können die anderen Microservices mit anderen Programmiersprachen entwickelt worden sein. Daher muss untersucht werden, welche Technologien existieren und welche sich davon am besten eignen.

Besonders wenn das Testkonzept zuerst auf den Import-Bereich angewendet wird, können die daraus gewonnenen Erfahrungen auf die anderen Bereiche umgesetzt werden. Wie eben gerade schon erwähnt, kann die Erstellung der Deployment Pipelines im Import-Bereich nützliche Informationen liefern und als Grundlage dienen.

Zu der Ausweitung des Testkonzeptes gehört auch, dass die Microservices getestet werden, die nur während der Simulation verwendet werden. Diese sind zurzeit vom Testkonzept ausgeschlossen. Um diese mit einschließen zu können, sind weitere Informationen und Untersuchungen notwendig. Daher kann auch kein Ausblick dafür getroffen werden

5.4.3 Anwendung auf ein allgemeines Microservice-basiertes System

Für das Anwenden dieses Testkonzeptes auf ein allgemeines Microservice-basiertes System, müssen verschiedene Sachen beachtet werden. Aufgrund der veralteten und fehlenden Dokumentation im MARS Framework ist ein starker Fokus im Testkonzept auf das Sammeln von

Informationen gelegt worden. Insbesondere für das Definieren von Anforderungen werden die Informationen benötigt. Dieses Sammeln von Informationen wird wahrscheinlich bei einem allgemeinen Microservice-basierten Systems nicht vonnöten sein. Es wird nämlich davon ausgegangen, dass solche Anwendungen Anforderungen an das System definiert haben.

Dadurch fällt die Verwendung von Exploratory Testing nicht weg. Es wird weiterhin gebraucht, um die Anwendung besser kennenzulernen und dabei zu unterstützen, End-To-End Tests zu entwickeln sowie zu verbessern.

Die im Testkonzept vorgestellten zeitlichen Grenzen bei den einzelnen Testebenen, müssen evaluiert werden, ob sie für ein allgemeines System passen. Zwar sind diese Zeiten vom Testkonzept von Google inspiriert worden, dennoch wurden die Zeiten so gewählt, wie sie am wahrscheinlichsten für das MARS Framework passen.

Besonders die Grenze von fünf End-To-End Tests für das MARS Framework muss sehr wahrscheinlich angepasst werden. Dabei kann keine allgemeingültige Aussage getroffen werden. Die Höchstanzahl von End-To-End Tests steht in starker Abhängigkeit zu der Anwendung, die getestet werden soll. Abwendungen wie z.B. Netflix benötigen wahrscheinlich mehr Tests auf dieser Ebene, während andere Anwendungen ggf. weniger benötigen. Diese Entscheidung muss von Anwendung zu Anwendung getroffen werden.

Auch die Wahl der verwendeten Technologien ist von Anwendung zu Anwendung neu zu entscheiden. Auch hier kann keine pauschale Aussage für alle Systeme getroffen werden, insbesondere da die Entscheidungen für die Technologien von Microservice zu Microservice stets neu gefällt werden müssen.

Eine weitere Schwierigkeit beim MARS Framework war der fortgeschrittene Status der Anwendung und dass sie weiterhin gravierende Änderungen erhält. Wie im Abschnitt 5.3.8 vorgestellt, kann das Testkonzept gut mit den verschiedenen Arten von Änderungen umgehen. Dazu ist das Testkonzept vom Status der Anwendung losgelöst, bis auf das Sammeln der Informationen für die Definition der Anforderungen. Aus diesen Gründen sollten an dieser Stelle keine Probleme entstehen oder Anpassungen notwendig sein, wenn das Testkonzept auf ein allgemeines System angewendet werden soll.

Dasselbe gilt auch für die Schwierigkeit bei der Kapselung von Aufrufen von Services. Es sind keine Informationen darüber bekannt, dass dies üblich oder unüblich für einen Microservice ist. Daher können die hier getroffenen Entscheidungen ggf. auch auf ein allgemeines System angewendet werden, wenn diese ähnlich wie im MARS Framework solche Aufrufe kapseln. Selbst wenn es das nicht macht, sind keine Anpassungen notwendig, weil dieser Aspekt dann einfach ignoriert werden kann.

5.5 Fazit

In diesem Kapitel wurde das Testkonzept für das MARS Framework vorgestellt. An das Testkonzept wurden mehrere Anforderungen gestellt. Zum einen müssen die bereits ausgearbeiteten Herausforderungen gemeistert werden und zum anderen muss das Testkonzept mit verschiedenen Arten von Änderung umgehen können.

Anhand dieser Anforderungen ist das Testkonzept entwickelt wurden. Das Testkonzept für das MARS Framework ist in zwei Bereiche unterteilt. Ein Bereich regelt das Testen eines einzelnen Microservices und der zweite Bereich das Testen des Gesamtsystems. In den einzelnen Bereichen existieren verschiedene Arten von Tests, die auf die einzelnen Anforderungen reagieren.

Neben der Vorstellung des Testkonzeptes wurde in diesem Kapitel ausführlich begründet, warum das Testkonzept so entworfen worden ist. Dazu wurde auf die einzelnen Ebenen und Testarten eingegangen. Dabei wurden auch Vor- und Nachteile der einzelnen Entscheidungen vorgestellt. Ergänzend dazu wurden auch die Informationen aus den vorgestellten Testkonzepten zur Begründung mit herangezogen.

In der Begründung für das Testkonzept wurden noch weitere Aspekte beleuchtet. Es wurde untersucht, wie das Testkonzept versucht, die einzelnen Herausforderungen zu lösen. Als Fazit kann hier gezogen werden, dass das Testkonzept für die meisten Herausforderungen eine Lösung gefunden hat. Allerdings wurden für die Herausforderung, die Microservices zu testen, die nur bei der Simulation Verwendung finden und die damit verbundene Herausforderung bezüglich der generischen Anforderungen keine Lösung gefunden. Dieser Aspekt wurden jedoch bei der Erstellung des Testkonzeptes ausgeklammert.

Als weiterer Punkt wurde vorgestellt, wie das Testkonzept mit den verschiedenen Arten von Änderungen umgeht. Dabei konnte gezeigt werden, dass das Testkonzept sehr gut mit den verschiedenen Arten von Änderungen umgehen kann.

Insgesamt kann also festgehalten werden, dass das hier vorgestellte Testkonzept seine Anforderungen erfüllt.

6 Future Work

Bei der Entwicklung des Testkonzeptes wurden viele verschiedene Aspekte sowie Bereiche vorgestellt und erläutert. Trotzdem sind einige Punkte offen geblieben, die in diesem Kapitel kurz beleuchtet werden sollen.

Bei der Vorstellung der Architektur Microservices erscheint immer wieder der Punkt, dass Microservices die Eigenschaft der Resilience besitzen sollen. Resilience beschreibt die Fähigkeit trotz fehlerhafter oder fehlender Abhängigkeiten weiterhin zu funktionieren. Im Umfeld von der Architektur Microservices ist dies z.B. der Fall, wenn ein Microservice ausfällt und seine Funktionen nicht mehr genutzt werden können, jedoch die von ihm abhängigen Microservices weiterhin funktionieren. Die Fragestellung ist hierbei, wie die Microservices getestet werden können, um zu beweisen, dass sie diese Eigenschaft besitzen. Dieser Aspekt wurde in dem hier vorgestellten Testkonzept nicht beleuchtet und gehört daher weiter untersucht.

Im hier entworfenen Testkonzept werden nur funktionale Anforderungen betrachtet. Ein umfassendes Testkonzept muss aber auch nicht-funktionale Anforderungen beachten. Dabei existiert die gleiche Schwierigkeit im MARS Framework wie bei den funktionalen Anforderungen, dass keine Anforderungen existieren. Dahingehend müssen mögliche Lösungen untersucht werden, wie dieses Problem behoben werden kann. Dabei kann Exploratory Testing wieder eine möglicher Weg sein, die Fragen zu klären.

Im Anschluss muss untersucht werden, wie nicht-funktionale Tests in einem Microservice Umfeld aussehen können. Dafür geben Newman und Wolff erste Ansätze in ihren Testkonzepten, die sich lohnen näher betrachtet zu werden [New16, Wol15].

Ein weiterer Aspekt der näher untersucht werden soll, ist spezifisch für das MARS Framework. Es geht dabei darum herauszufinden, wie die Microservices, die nur in der Simulation verwendet werden, getestet werden können. Dieser Aspekt wurde in dieser Arbeit nicht untersucht, da er zum umfangreich ist. Trotzdem haben diese Microservices keinen geringen Anteil an der Gesamtanzahl der Microservices ab. Sie bilden im Gegenteil die Kernaufgabe des MARS Frameworks ab und zwar die Multi-Agenten Simulation. Daher muss untersucht werden, wie diese Microservices effizient getestet werden können, ohne den Aufwand und die Dauer einer kompletten Simulation zu haben. Dazu gehört auch die Fragestellung, was die Anforderung

beim Testen dieser Microservices ist. In diesem Zusammenhang stellt sich die Frage, an welchen Kriterien festgestellt werden kann, ob ein Test erfolgreich ist.

Das hier entwickelte Testkonzept für das MARS Framework ist bisher nur ein theoretischer Ansatz. Es wurde in die Praxis noch nicht umgesetzt. Daher ist es eine der nächsten Aufgaben, dieses Testkonzept in der Praxis anzuwenden. Es soll dabei untersucht werden, ob das Testkonzept in die Praxis umsetzbar ist.

Wenn die Umsetzung in die Praxis erfolgt, muss vor allem geprüft werden, ob die benannten Herausforderungen mit den hier gegebenen Antworten gelöst werden können. Ein praktisches Anwenden dieses Konzeptes kann nur in enger Abstimmung mit der MARS-Gruppe geschehen, denn das Testkonzept kann nur erfolgreich sein, wenn es von den Entwicklern auch akzeptiert wird.

7 Gesamtfazit

Das primäre Ziel dieser Arbeit war ein Testkonzept für das MARS Framework zu entwickeln. Dazu sollte, wie in der Einleitung erwähnt, die Wissenslücke zum Testen von Microservice-basierten Systemen geschlossen werden. Für die Entwicklung eines solchen Testkonzeptes und die Schließung der Wissenslücke wurden verschiedene Leitfragen aufgestellt, um diesen Prozess zu unterstützen.

Eine grundlegende Leitfrage war dabei, zu erkunden, was Microservices eigentlich sind. Diese Frage konnte im Grundlagenkapitel (Abschnitt 3.2) zu Microservices beantwortet werden. Dabei wurden die verschiedenen Charakteristika, wie z.B. die starke Modularisierung und die lose Kopplung, näher vorgestellt.

Dazu ergänzend wurden auch die besonderen Vorteile der Architektur Microservices präsentiert und erläutert. Der größte Vorteil entsteht durch die starke Modularisierung. Dadurch werden die Microservices unabhängig voneinander. Dieser Umstand besitzt Einfluss auf den typischen Build Prozess einer Microservice-basierten Anwendung. Aufgrund dieses Umstandes kann jeder Microservice seine eigene Deployment Pipeline verwenden. Hierdurch können die Deployment Pipelines optimal an ihre Microservices angepasst werden. Mit diesen beiden Abschnitten wurde die zweite Leitfrage nach den Besonderheiten beantwortet.

Bei der nächsten Leitfrage ging es darum, welche Herausforderungen beim Testen einer Anwendung mit Microservices vorhanden sind. Diese Frage war zweigeteilt und zwar einmal nach den allgemeinen Herausforderungen und zum anderen nach den Herausforderungen, die vom MARS Framework kommen. Da ein Testkonzept für das MARS Framework entwickelt werden sollte, war es wichtig zu ergründen, ob spezifische Herausforderungen existieren, und wenn ja, welche dies sind.

Bei der Ausarbeitung dieser Herausforderung ging es darum, die Schwierigkeiten zu erkennen, mit denen das Testkonzept umgehen muss. Dahingehend wurde dem Testkonzept auch vorgegeben, auf die gefundenen Herausforderungen eine Antwort zu finden.

Eine der größten Herausforderungen für das Testkonzept war der Umgang mit Änderungen. Gerade die einfache Handhabung von Änderungen bildet einen großen Vorteil dieser Architek-

tur, deswegen muss das Testkonzept dies unterstützen. Da dies ein sehr wichtiger Aspekt war, wurde dafür eine eigene Leitfrage definiert.

Neben dem Umgang mit Änderungen, wurde in der Leitfrage auch die Thematik behandelt, ob verschiedene Arten von Änderungen existieren. Bei näherer Betrachtung der Architektur Microservices konnte dies festgestellt werden. Anhand dieses Ergebnisses wurden die verschiedenen Arten von Änderungen klassifiziert. Insgesamt wurden sechs verschiedene Klassen von Änderungen definiert. Dem Testkonzept wurde als weitere Anforderung vorgegeben, mit den verschiedenen Klassen von Änderungen umgehen zu können.

Die letzte Leitfrage definierte das primäre Ziel dieser Masterarbeit detaillierter und zwar die Entwicklung eines Testkonzeptes für das MARS Framework. Dabei soll das Testkonzept die verschiedenen Anforderungen erfüllen. Zusätzlich sollte es aufschlüsseln, welche Teststufen existieren und wie sie definiert werden.

Um dafür mögliche Lösungen zu finden, wurden vier verschiedene bereits existierende Testkonzepte untersucht. Drei von diesen vier Testkonzepten sind explizit auf das Testen eines Microservice-basierten Systems ausgelegt. Neben der Präsentation der einzelnen Testkonzepte wurden sie auch vergleichend gegenüberstellt. Dabei wurden besonders die einzelnen Teststufen in Hinsicht auf Gemeinsamkeiten und Unterschiede miteinander verglichen. Zum Abschluss wurden die einzelnen Testkonzepte noch hinsichtlich ihrer Zweckmäßigkeit bewertet. Hierbei konnte ermittelt werden, dass kein Testkonzept weder sehr gut noch sehr schlecht war. Alle Testkonzepte besitzen ihre Vor- und Nachteile.

Ergänzend dazu wurde die Methode Exploratory Testing betrachtet. Dafür wurde diese Methode in mehreren Szenarien mit automatisierten Tests verglichen. Als Ergebnis wurde festgehalten, dass sich Exploratory Testing insbesondere beim Testen von neuen Features eignet.

Auf Grundlage der beschriebenen Leitfragen und die dazugehörige Leitfragen für die Erstellung des Testkonzeptes wurde das Testkonzept für das MARS Framework entwickelt. Es unterteilt sich dabei in zwei Ebenen. Eine Ebene ist für das Testen eines einzelnen Microservices gedacht und die andere Ebene für das Testen des Gesamtsystems. Die einzelnen Ebenen besitzen dabei unterschiedliche Teststufen.

Im Anschluss an die Präsentation des Testkonzeptes wurde ausführlich erläutert, warum das Testkonzept so konzipiert wurde. Ergänzend dazu wurde auch beleuchtet, wie das Testkonzept mit den einzelnen Besonderheiten und Herausforderungen der Architektur Microservices umgeht.

Hier kann als Ergebnis festgehalten werden, dass das Testkonzept mit den Besonderheiten der Architektur Microservices in ausreichendem Maße umgehen kann und auf die meisten

Herausforderungen eine Lösung gefunden hat. Auf die nicht gemeisterten Herausforderungen wurde im Kapitel 6 Future Work näher eingegangen. Diese offenen Herausforderungen bilden die Grundlage für die weiteren Untersuchungen.

Dennoch kann schlussendlich das Fazit gezogen werden, dass es innerhalb dieser Arbeit gelungen ist, ein Testkonzept für das MARS Framework zu entwerfen. Dieses Testkonzept ist dabei an die Architektur Microservices und an das MARS Framework angepasst, so dass die Vorteile der Architektur unterstützt werden und die meisten Herausforderungen gemeistert werden.

Literaturverzeichnis

[AS12] Tilo Linz Andreas Spillner. *Basiswissen Softwaretest.* Dpunkt.Verlag GmbH, 2012.

[BS15] Tobias Bayer and Hendrik Still. Microservices: Consumer-driven Contract Testing mit Pact, 2015. Zugriff am 29.12.2017 https://jaxenter.de/microservices-consumer-driven-contract-testing-mit-pact-20574.

[Bur12] David Burns. *Selenium 2 Testing Tools.* Packt Publishing, 2012.

[Cle14] Toby Clemson. Testing strategies in a microservice architecture, November 2014. Zugriff am 30.01.2017 https://martinfowler.com/articles/microservice-testing/.

[Coh09] Mike Cohn. *Succeeding with Agile.* Addison Wesley, 2009.

[Fie14] Jay Fields. *Working Effectively with Unit Tests.* CreateSpace Independent Publishing Platform, 2014.

[FL15] Martin Fowler and James Lewis. Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr. *Objektspektrum*, 1(2015):14–20, 2015.

[Fow] Fowler. Who Has Used Them? Zugriff am 013.04.2017 https://www.martinfowler.com/microservices.

[Fow11] M Fowler. Eradicating non-determinism in tests, 2011. Zugriff am 01.06.2017 https://martinfowler.com/articles/nonDeterminism.html.

[Fow12] Martin Fowler. Testpyramid, May 2012. Zugriff am 11.06.2017 https://martinfowler.com/bliki/TestPyramid.html.

[Fow14] Martin Fowler. Microservices. http://martinfowler.com/articles/microservices.html, 2014. Zugriff am 23.05.2016.

[Fow16] Susan J. Fowler. *Production-Ready Microservices*. O'Reilly UK Ltd., 2016.

[Gun15] Unmesh Gundecha. *Selenium Testing Tools Cookbook Second Edition*. PACKT PUB, 2015.

[HATCea16] Christian Huening, Mitja Adebahr, Thomas Thiel-Clemen, and et al., editors. *Modeling & Simulation as a Service with the Massive Multi-Agent System MARS*. To appear in Proceedings of the 2016 Spring Simulation Multiconference, 2016.

[Hom13] Bernard Homès. *Fundamentals of Software Testing*. John Wiley & Sons, 2013.

[HSS16] Michael Hofmann, Erin Schnabel, and Katherine Stanley. *Microservices Best Practices for Java*. IBM Redbooks, 2016.

[IR05] J. Itkonen and K. Rautiainen. Exploratory testing: a multiple case study. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10 pp.–, Nov 2005.

[JH10] David Farley Jez Humble. *Continuous Delivery*. Pearson Technology Group, 2010.

[mar] MARS Group HAW Hamburg. http://mars-group.mars.haw-hamburg.de/. Zugriff am 30.05.2017.

[New15] Sam Newman. *Microservices (mitp Professional)*. MITP Verlags GmbH & Co. KG, 2015.

[New16] Sam Newmann. *Building Microservices*. O'Reilly UK Ltd., first edition edition, 2016.

[O'H06] Charlene O'Hanlon. A conversation with Werner Vogels. *Queue*, 4(4):14:14–14:22, May 2006.

[pac17] Pact, January 2017. Zugriff am 30.01.2017 https://docs.pact.io/.

[Pie16] Alexander Piehl. Testen von Microservices. 2016. Ausarbeitung Hauptseminar.

[Pie17a] Alexander Piehl. Consumer Driven Contract Tests. 2017. Ausarbeitung Hauptprojekt.

[Pie17b] Alexander Piehl. Das Testen von Microservices. 2017. Ausarbeitung Grundprojekt.

[SBF14] IEEE Computer Society, Pierre Bourque, and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.

[sel] Selenium HQ. http://www.seleniumhq.org/. Zugriff am 24.10.2017.

[SR15] Dmitry Savchenko and Gleb Radchenko. Microservices validation: Methodology and implementation. In *EUR Workshop Proceedings. Vol. 1513: Proceedings of the 1st Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists (Ural-PDC 2015).—Yekaterinburg, 2015.*, 2015.

[SRT15] D. I. Savchenko, G. I. Radchenko, and O. Taipale. Microservices validation: Mjolnirr platform case study. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 235–240, May 2015.

[Vau97] Diane Vaughan. *The Challenger launch decision: Risky technology, culture, and deviance at NASA*. University of Chicago Press, 1997.

[WAC12] James A Whittaker, Jason Arbon, and Jeff Carollo. *How Google tests software*. Addison-Wesley, 2012.

[Whi09] James A. Whittaker. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Addison-Wesley Professional, 3th edition, 2009.

[Wol15] Eberhard Wolff. *Microservices: Grundlagen flexibler Softwarearchitekturen*. Dpunkt. verlag, erste auflage edition, 2015.

[WSH16] Eberhard Wolff, Alexander Schwartz, and Alexander Heusingfeld. *Microservices: Der Hype im Realitaetscheck*. entwickler.press, 2016.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 11. Januar 2018 Alexander Piehl