Hochschule Ravensburg-Weingarten
University of Applied Sciences

Automatisierte Teststrategien für Microservice-Architekturen

Bachelorarbeit

im Studiengang Angewandte Informatik

Verfasser: Jan Eisenhauer
Am Dorfstadel 39
D-88410 Bad Wurzach
Matrikelnummer: 29991

1. Betreuer: Prof. Dr. rer. nat. Marius Hofmeister
2. Betreuer: Prof. Dr. Sebastian Mauser

Bearbeitungszeitraum: 2. September 2021 bis 29. März 2022
Abgabedatum: 29. März 2022

Die moderne Welt fordert stetig schnellere Entwicklungen von Softwareprojekten, um neue Innovationen frühzeitig an den Markt zu bringen. Um diesen Ansprüchen gerecht zu werden, werden häufiger Microservice-Architekturen eingesetzt. Eine Microservice-Architektur teilt ein System in kleine Komponenten ein, die unabhängig entwickelt werden. Durch die Unabhängigkeit kann die Bereitstellung der Software automatisiert und neue Anforderungen an einzelne Microservices zügiger umgesetzt werden.

Bei der raschen Implementierungen soll die Software-Qualität nicht vernachlässigt werden. Deswegen stellt sich die Frage, wie sich die Servicequalität einer Microservice-Architektur sicherstellen lässt. Das Ziel dieser Arbeit ist es, automatisierte Teststrategien für Microservice-Architekturen herauszuarbeiten, mit denen die Software-Qualität gesteigert wird und Fehler in der Software frühzeitig erkannt werden.

Um das Ziel zu erreichen, wurden bekannte Teststrategien aus der Literaturen untersucht und auf eine Microservice-Architektur angewendet. Dazu gehören Unit-Tests, welche die kleinsten Einheiten prüfen, Integrationstests, Komponententests und Contract-Tests, die die Interaktionen zwischen den Microservices testen, und End-To-End-Tests, die schlussendlich die Funktionalität des Gesamtsystems kontrollieren. Dabei wurden die verschiedenen Teststrategien vorgestellt und anschließend in einer Beispielanwendung als konkrete Implementierung vorgeführt und bewertet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface. 16, 20–22, 24, 53–55

CI/CD Continuous-Integration / Continuous-Delivery. 14, 21, 56

HTML HyperText Markup Language. 46, 47

HTTP Hypertext Transfer Protocol. 19, 39, 43

ISTQB International Software Testing Qualifications Board. 3

JPA Java Persistence API. ii, 30, 34, 55

JSON JavaScript Object Notation. 37, 38, 40, 45

ORM Object-relational mapping. 30

REST Representational State Transfer. v, 10, 22, 24, 30, 37, 39, 40, 43, 54, 55

RPC Remote Procedure Call. 10

UI User-Interface. i, ii, v, 14–17, 22, 23, 46–49, 53, 54

UML Unified Modeling Language. 6

URL Uniform Resource Locator. 34, 38, 43

Listingverzeichnis

1 Einleitung

In diesem Kapitel wird das Thema dieser Arbeit erläutert. In Abschnitt 1.1 werden die Beweggründe für automatisierte Teststrategien für Microservice-Architekturen verdeutlicht. Anschließend wird in Abschnitt 1.2 auf die Ziele dieser Arbeit eingegangen und in Abschnitt 1.3 der Aufbau der Arbeit dargestellt.

1.1 Motivation

In den letzten Jahren hat ein Umdenken in der Software-Architektur stattgefunden. Viele Unternehmen wie Zalando, Netflix und Amazon stellten ihre monolithischen Software-Strukturen auf eine Microservice-basierte Lösung um [3, 4].

Microservices sind ein Architektur-Ansatz, bei welchem eine Anwendung in viele einzelne Dienste (Microservice) unterteilt wird. Diese Dienste laufen jeweils in einem eigenen Prozess oder in einer eigenen Umgebung und sind stark voneinander entkoppelt. Zur Kommunikation zwischen den einzelnen Diensten wird häufig ein Request-Response-Verfahren oder eine Event-gesteuerte Architektur verwendet. Microservices bieten den Vorteil einer starken Modularisierung der Anwendung, wodurch die Strukturierung der gesamten Software erleichtert wird. Da die Dienste unabhängig voneinander sind, lässt sich ein einzelner Dienst in einem kleineren Team mit einer agilen Arbeitsweise entwickeln. Ebenso vereinfacht sich die Wartbarkeit, das Deployment und die Skalierbarkeit der Anwendung durch die einzelnen Microservices [5].

Bereits in monolithischen Anwendungen hat sich die Entwicklung von automatisierten Tests bewährt, um Anforderungen und Qualitätsstandards an die Software zu erfüllen und um Mängel und Fehler in der Anwendung frühzeitig zu erkennen [6]. Die höhere Komplexität einer Microservice-Architektur erfordert dabei jedoch neue Herangehensweisen, um die Anwendung im Ganzen testen zu können.

Daher stellt sich die Frage, wie lässt sich die Servicequalität von Microservice-Architekturen mithilfe von automatisierten Teststrategien sicherstellen und verbessern, um den immer komplexer werdenden Anforderungen an die Microservices gerecht zu werden.

1.2 Zielsetzung

Das Ziel der Arbeit ist es, bekannte Teststrategien aus der Literatur aufzugreifen und auf eine Microservice-Architektur zu übertragen und anzuwenden. Dabei werden die Teststrategien in Bezug auf die Verbesserung der Servicequalität und der Realisierbarkeit bewertet und Unterschiede zu monolithischen Anwendungen dargestellt. Des Weiteren werden neue Teststrategien erarbeitet und untersucht, welche sich speziell für Microservice-Architekturen eignen. Die Teststrategien werden anschließend anhand einer Beispielanwendung implementiert, um eine konkrete Umsetzung der Teststrategien aufzuzeigen.

1.3 Überblick über die Arbeit

Diese Arbeit ist in fünf Kapiteln eingeteilt. Kapitel 1 stellt das Thema dieser Arbeit vor und vermittelt einen Überblick über die Thematik. In Kapitel 2 werden Grundlagen zu Softwaretests und Software-Architekturen vorgestellt und die Microservice-Architektur näher erklärt. Kapitel 3 befasst sich mit den verschiedene Teststrategien, welche für Microservice-Architekturen eingesetzt werden können. In Kapitel 4 werden die vorgestellten Teststrategien für Microservice-Architekturen aufgegriffen und auf ein beispielhaftes Projekt angewendet. Schlussendlich fasst Kapitel 5 den Inhalt dieser Arbeit zusammen und gibt einen Ausblick über weiterführende Fragestellungen.

2 Grundlagen

Im folgenden Kapitel wird grundlegendes Wissen für diese Arbeit vermittelt. In Abschnitt 2.1 werden Softwaretests erklärt. Anschließend leitet der Abschnitt 2.2 in das Thema der Software-Architekturen ein, in welchem die monolithische Architektur und die Microservice-Architektur vorgestellt werden. Danach führt Abschnitt 2.3 näher in die Thematik der Microservices ein.

2.1 Softwaretests

Softwaretests sind eine Maßnahme in der analytischen Qualitätssicherung, um die Qualität einer Software zu verbessern und zu steigern. Dabei wird die zu untersuchende Software oder ein Programmteil mit definierten Eingabedaten ausgeführt und anschließend die zurückgelieferten Ergebnisse des Programms mit den Soll-Ergebnissen verglichen. Das Ziel von Softwaretests ist das frühzeitige Entdecken von Fehlern in der Software.

> „An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.“ [7]

Das International Software Testing Qualifications Board, bekannt unter dem Akronym ISTQB, bietet eine standardisierte Ausbildung und Zertifizierung für Softwaretester an. Diese teilt den Testprozess im Testmanagement in die Aktivitäten Planung und Steuerung, Analyse und Design, Realisierung und Durchführung, Auswertung und Bericht, und Abschluss ein [8].

2.1.1 Prüfebene

In der Entwicklungsphase lassen sich Softwaretests in verschiedenen Prüfebenen zuordnen. Zu diesen Prüfebenen gehören Unit-Tests, Integrationstests und Systemtests. [9]

In der ersten Prüfebene befinden sich die Unit-Tests, welche eine atomare Programmeinheit testen. Ein Unit-Test kann eine einzelne Funktion, aber auch eine Klasse oder

ein Modul, einer Software umfassen. Unit-Tests werden sehr nahe am entsprechenden Quellcode entwickelt und zielen darauf ab, eine möglichst kleine Einheit zu testen.

Auf der zweiten Prüfebene stehen die Integrationstests, welche dann eingesetzt werden, wenn Software- oder Programmmodule zu größeren Komponenten zusammengesetzt werden und miteinander interagieren. Dadurch wird sichergestellt, dass diese Interaktion zwischen den Komponenten fehlerfrei abläuft.

Auf der dritten Prüfebene halten sich die Systemtests auf. Ein Systemtest wird auf das Gesamtsystem angewendet und testet eine Anwendung aus einer funktionalen Sichtweise. Dazu wird die gesamte Anwendung in einem möglichst zur Produktivumgebung ähnlichen Umfeld gestartet und darauf die Systemtests ausgeführt. Dadurch wird die Funktionalität des Gesamtsystems gewährleistet.

Die letzte Prüfebene beinhaltet die Abnahmetests, in welchen die Kunden miteinbezogen werden und somit eine Überprüfung der Gesamtanwendung in der realen Einsatzumgebung stattfindet. Abnahmetests geschehen in der Regel manuell, daher werden Abnahmetests nicht in dieser Arbeit behandelt.

2.1.2 Prüfkriterien

Neben der Zuordnung von Softwaretests in Prüfebenen lassen sich Softwaretests auch in Prüfkriterien klassifiziert, welche die inhaltlichen Aspekte der Anwendung prüfen. Diese Prüfkriterien sind in die drei Kategorien funktionale, operationale und temporale Softwaretests eingeteilt. [9]

Funktionale Softwaretests kontrollieren eine Anwendung auf seine Funktionalität, wobei das System bei gegebenen Eingangsgrößen die richtigen Ausgangsgrößen berechnet. Hierzu zählen auch Crashtests, die Abstürze der Software provozieren, und Kompatibilitätstests, welche die Ausführbarkeit der Software auf verschiedenen Hardware- und Softwareplattformen untersuchen.

Operationale Softwaretests hingegen prüfen die operativen Aspekte wie die Inbetriebnahme oder die Benutzbarkeit der Software und kümmern sich zudem um die Vermeidung von Sicherheitslücken sowie die Vermeidung von Gefahren, welche von der Anwendung ausgehen können.

In temporalen Softwaretests werden Laufzeittests, Lasttests und Stresstests durchgeführt, um die Software auf ihre Zeitanforderungen und das Verhalten der Anwendung bei Extremsituationen zu prüfen.

Operationale und temporale Softwaretests werden häufig zusammengefasst und als nicht

funktionale Softwaretests bezeichnet.

2.1.3 Prüftechnik

Mit der Konstruktion von Testfällen beschäftigt sich die Prüftechnik beziehungsweise die Prüfmethodik. Die Konstruktion wird in Black-Box-Tests, White-Box-Tests und Gray-Box-Tests unterschieden [9].

Bei Black-Box-Tests wird lediglich das gewünschte Eingabe- und Ausgabeverhalten beachtet, welches aus der Anforderungsbeschreibung für die Anwendung abgeleitet wird. Der Quellcode wird bei Black-Box-Tests nicht berücksichtigt. Dieses Vorgehen ermöglicht unter anderem testgetriebene Entwicklung, bei der zuerst die Tests für einen Programmteil programmiert werden, bevor die eigentliche Implementierung stattfindet.

White-Box-Tests hingegen analysieren die innere Struktur des Programm-Quellcodes und leiten daraus verschiedene Testfälle ab. Dabei wird zwischen kontrollflussorientierten Strukturtests und datenflussorientierten Strukturtests unterschieden. Bei der Kontrollflussmodellierung wird ein Kontrollflussgraph aus dem imperativem Programmierparadigma abgeleitet. Bei der Datenflussmodellierung geschieht die Ableitung anhand der internen Daten, wie zum Beispiel die Zuweisung einer Variable. Anhand dieser Strukturierung können mit verschiedenen Methoden konkrete Testfälle abgeleitet werden. Das Ziel hierbei ist, eine möglichst hohe Testabdeckung mit möglichst wenigen Testfällen zu erreichen. Zum Beispiel können bestimmte Testfälle ausgewählt werden, welche alle Pfade und Zweige innerhalb der aufgerufenen Funktion durchlaufen. [9]

Gray-Box-Tests werden zunächst nach dem Prinzip der Black-Box-Tests konstruiert, nehmen jedoch Einflüsse durch die innere Struktur des Programms. Diese Einflüsse stammen zum Beispiel daher, dass der Entwickler der Tests bereits die Implementierungsdetails des zu testenden Programmteils kennt.

2.2 Softwarearchitekturen

In der Informatik beschreibt die Softwarearchitektur die Struktur, den Aufbau und die Organisation des Gesamtsystems einer Software. Dabei definiert das Gesamtsystem alle Komponenten, wie diese untereinander interagieren und in welcher Umgebung diese sich befinden. Zudem gibt die Softwarearchitektur Prinzipien für den Entwurf und die Gestaltung des Softwaresystems an.

> „The fundamental organization of a system embodied in its components, their

> relationships to each other, and to the environment, and the principles guiding its design and evolution.“ [10]

In der Softwarearchitektur erfolgt zunächst die statische Zerlegung des Systems, wobei das System in seine physischen Bestandteile, den einzelnen Komponenten, aufgeteilt wird. Die Zerlegung bildet damit die Struktur der Softwarearchitektur ab. Das dynamische Zusammenwirken aller Komponenten beschreibt die Interaktionen zwischen den Komponenten und das Verhalten des Systems. Die Dynamik beziehungsweise das Verhalten des Systems baut auf die Softwarestruktur auf und definiert, wie die Komponenten über ihre Schnittstellen zusammenarbeiten. Die Strategie für die Softwarearchitektur gibt dabei an, wie das Zusammenspiel zwischen der Statik und der Dynamik funktionieren soll. [11]

Das Ziel einer Softwarearchitektur ist die Sicherstellung, dass funktionale sowie nicht funktionale Anforderungen erfüllt werden können. Die Softwarearchitektur stellt eine Abstraktion der konkreten Implementierung dar, beschreibt technische Lösungen für die Anforderungen an das System und veranschaulicht somit die Struktur und den Aufbau des Gesamtsystems. Gerade für Entwickler dient die Abstraktion zur Übersicht und trägt zum Verständnis über das System bei. Ebenfalls dient die Softwarearchitektur als Basis für die Überwachung des Projektfortschritts. Die Softwarearchitektur wird häufig inkrementell und iterativ aufgebaut und mit einer agilen Vorgehensweise erarbeitet. Dadurch können Erfahrungen, die in der konkreten Implementierung und Umsetzung gemacht werden, in die Architektur der Software miteinfließen. Zu Beginn eines Projektes kann ein Prototyp zur Verifizierung der Softwarearchitektur zur Hilfe genommen werden, um erste Ansätze in der Architektur zu bestätigen. [11]

Während in der Softwarearchitektur der Grundstein für die funktionalen Anforderungen an das zu realisierende System gelegt wird, bestimmt die Architektur ebenso Anforderungen an die Qualität der Software. Zu den nicht funktionalen Qualitätsanforderungen zählen unter anderem Modifizierbarkeit, Wartbarkeit, Sicherheit und Performance des Softwaresystems. Für die Umsetzung dieser Qualitätsanforderungen wird eine hohe Kohäsion innerhalb der Komponenten und eine möglichst schwache Kopplung der Komponenten zueinander angestrebt.[11]

Um die Softwarearchitektur zu visualisieren und zu dokumentieren, wird oft die Unified Modeling Language (UML) verwendet. Diese bietet verschiedene Notationen für Softwaremodellierung an. Zum Beispiel kann ein Komponentendiagramm für die Softwarearchitektur erstellt werden, um die Komponenten des Systems und deren Beziehungen zueinander grafisch darzustellen.

Zur Veranschaulichung teilt Kruchten in seinem Beitrag „The 4+1 View Model of architecture“ die Softwarearchitektur in fünf Sichtweisen für verschiedene Interessengruppen ein [1]. Das Ansichtsmodell ist in Abb. 2.1 grafisch dargestellt. Folgende Sichtweisen stellt

Kruchten vor:

Logical view: Die logische Ansicht fokussiert sich auf die Funktionalitäten, welche das System an den Endanwender zur Verfügung stellt. Dabei wird das System mit einem objektorientierten Ansatz in seine Komponenten aufgeteilt.

Process view: Die Prozessansicht baut auf die logische Ansicht auf und bezieht nicht funktionale Anforderungen wie die Performance und die Verfügbarkeit des Systems mit ein. Dazu werden zusammenhängende Aufgaben im System in einzelne ausführbare Einheiten gruppiert und anschließend bestimmt, wie diese aufgeteilten Aufgaben miteinander kommunizieren. Die Kommunikation kann synchron oder asynchron erfolgen. Mit der Prozessansicht kann die Verteilung des Systems und die Nebenläufigkeit der Aufgaben klar dargestellt werden. Auch wird die Systemintegrität und die nötige Fehlertoleranz berücksichtigt.

Development view: Die Entwicklersicht organisiert das System und deren Komponenten in Softwaremodule in der Entwicklungsumgebung. Die Software wird in kleine programmierbare Teile aufgeteilt, um somit das Projektmanagement zu unterstützen.

Physical view: In der physischen Ansicht wird die Software auf die Hardware abgebildet. Sie gibt an, auf welchem Computer oder Server die Software tatsächlich ausgeliefert und ausgeführt wird. Zum Beispiel kann die Software für die Entwicklung und das Testen auf eine entsprechende Entwicklungs- oder Testumgebung mit bestimmten Servern bereitgestellt werden.

Scenarios: In der Szenarioansicht werden die vier vorherigen Ansichten zusammengeführt und Anwendungsfälle modelliert. Mit den Anwendungsfällen können erste Prototypen für die Architektur entworfen und Elemente in der Architektur leichter gefunden werden.

2.2.1 Monolithische Architektur

Bei einer monolithischen Architektur wird die gesamte Anwendung als eine zusammenhängende Software entwickelt. Die vollständige Software wird als ein gebündeltes Programm ausgeliefert und bereitgestellt. Beim Ausführen des Programms läuft die gesamte Software in einem einzigen Prozess. Da die Software zu einem Programm gebündelt wird, geschieht die Implementierung des gesamten Monolithen in einer einzigen Programmiersprache. Die monolithische Softwarearchitektur ist innerhalb des Programms in verschiedene Komponenten aufgeteilt, die unmittelbar miteinander interagieren können. Diese Interaktion kann im Quelltext beispielsweise ein Funktionsaufruf auf eine andere Komponente sein.

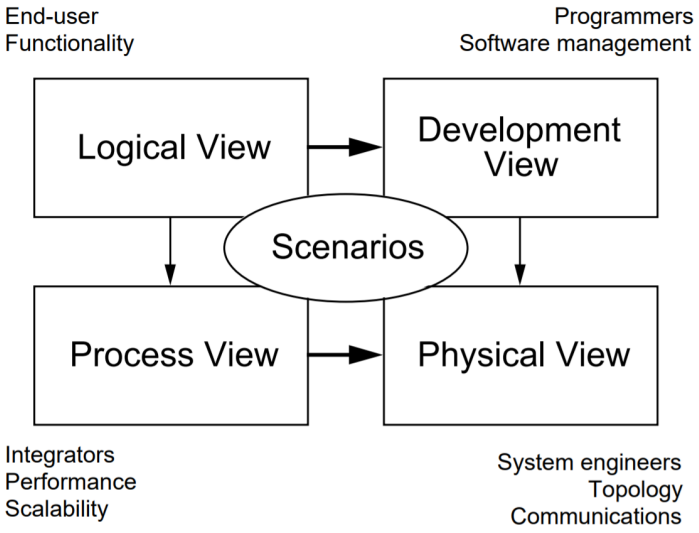

Abbildung 2.1: 4+1 Ansichtsmodell von Kruchten [1]

Um die Anwendung zu skalieren, kann das Programm mehrmals auf derselben oder auch auf verschiedenen Plattformen ausgeführt werden. Bei diesem Vorgehen handelt es sich um einen verteilten Monolithen, wobei Aspekte eines verteilten Systems beachtet werden müssen [12]. Zu diesen Aspekten gehört zum Beispiel der richtige Umgang von Zuständen der einzelnen Programmprozesse und wie diese zwischen den einzelnen Instanzen des Programms kommuniziert und ausgetauscht werden.

Die monolithische Architektur ist die am häufigsten verwendete Architektur und der de facto Standard bei neuen Softwareprojekten. Die monolithische Architektur kommt dabei sowohl bei kleineren als auch bei größeren Projekten zum Einsatz.

2.2.2 Microservice-Architektur

Im Gegensatz zur monolithischen Architektur ist die Microservice-Architektur eine serviceorientierte Architektur, die auf eine unabhängige Bereitstellung der einzelnen Komponenten abzielt. Eine einzelne Komponente wird meist um eine Geschäftslogik herum modelliert und bildet somit einen einzelnen Microservice. Die Microservices werden unabhängig voneinander entwickelt, wobei sie ihre Implementierung anderen Microservices gegenüber verstecken. Dadurch können andere Technologien für jedes Microservice eingesetzt werden. Zum Beispiel können verschiedene Programmiersprachen für unterschiedliche Microservices verwendet werden. Aufgrund der Unabhängigkeit verwendet jedes Microservice in der Regel auch seine eigene Datenbank, ohne diese mit anderen Microservices zu teilen. Die Kommunikation zwischen den Microservices erfolgt über Netzwerk-Schnittpunkte, die ein Microservice für andere Microservices anbieten kann.

Alle Microservices zusammen bilden so das Gesamtsystem der Anwendung. [12]

Da die Microservices unabhängig voneinander agieren, kann jedes einzelne Microservice separat skaliert werden. So können gezielt mehrere Instanzen eines Microservices gestartet werden, die gerade eine hohe Last erfahren. Dadurch lässt sich die Anwendung auf Ebene der Komponenten beliebig skalieren.

Die Microservice-Architektur kommt oft dann zum Einsatz, wenn die Anwendung eine gewisse Komplexität übersteigt. Bei einem neuen Softwareprojekt wird selten zu Beginn direkt eine Microservice-Architektur implementiert. Stattdessen erfolgt die Implementierung der Microservices aus einer etablierten, monolithischen Anwendung heraus. Dabei wird der Monolith nach und nach aufgeteilt und Komponenten innerhalb des Monolithen durch Microservices ersetzt.

2.3 Microservices

Microservices sind ein serviceorientierter Softwarearchitekturansatz. Das Gesamtsystem wird in unabhängige Dienste (Microservices) unterteilt, die in isolierten Prozessen ausgeführt werden. Über Netzwerk-Schnittstellen können diese Dienste miteinander kommunizieren. So entsteht eine modulare Anwendungssoftware, in welcher die Komponenten lose gekoppelt sind.

Ein zentrales Element der Microservice-Architektur ist die Unabhängigkeit der einzelnen Microservices. Von der Entwicklung über das Bauen des Programms bis hin zur Bereitstellung eines Microservices in die Produktionsumgebung zielt die Microservice-Architektur auf eine lose Kopplung der Komponenten ab. Durch die Unabhängigkeit können die verwendeten Technologien für ein einzelnes Microservice an die Ansprüche an dieses Microservice angepasst werden.

Nicht nur die Microservices selbst werden unabhängig voneinander erstellt, sondern auch der Zustand der Microservices, wie zum Beispiel abgespeicherte Daten, wird unabhängig angelegt. In der Regel erhält deswegen jedes Microservice seinen eigenen Datenspeicher. Dadurch kann für jedes Microservice der optimale Datenspeicher eingesetzt werden. Das Ziel hierbei ist das Verstecken der Informationen nach außen, sodass das Microservice volle Kontrolle über seine Daten behält.

Die Abgrenzungen der Microservices geschieht meist entlang von Geschäftsbereichen. Häufig wird für die Entwicklung von Microservices auf Domain-Driven Design zurückgegriffen und die Microservices anhand von Event Storming, eine Brainstorming-Methode, bestimmt und abgegrenzt [13].

Wie der Name Microservices schon andeutet, sind die einzelnen Microservices klein gehalten, um ein einzelnes Microservices übersichtlich zu gestalten. Dadurch ist es möglich, dass alle Microservices auf verschiedene kleine Teams aufgeteilt werden und ein Microservice jeweils von einem kleinen Team verwaltet wird.

> „A microservice should be as big as your head.“[12]

2.3.1 Kommunikation

Im Gegensatz zu einem Monolithen, in dem Funktionen von anderen Komponenten im selben Prozess aufgerufen werden können, geschieht die Kommunikation zwischen den Microservices über ein Netzwerk.

Bei der Kommunikation von Microservices wird zwischen einer synchronen und asynchronen Kommunikation unterschieden. Bei einer synchronen Kommunikation stellt ein Microservice eine Anfrage über das Netzwerk an ein anderes Microservice und wartet im Prozess auf dessen Antwort. Eine asynchrone Anfrage wiederum fährt mit anderen unabhängigen Aufgaben fort und reagiert erst bei einer Antwort auf die zuvor gestellte Anfrage.

Die synchrone Kommunikation wird meistens mit einem Request-Response-Verfahren realisiert. Mithilfe von zum Beispiel Representational State Transfer (REST) oder Remote Procedure Call (RPC) wird eine Anfrage an einen untergeordneten Microservice direkt gestellt. Der Prozessablauf wird dabei so lange blockiert, bis der untergeordnete Microservice eine Antwort ausgeliefert hat. Das Request-Response-Verfahren kann auch bei einer asynchronen Kommunikation verwendet werden, mit dem Unterschied, dass der Prozessablauf nicht blockiert wird.

Alternativ kann eine asynchrone Kommunikation zwischen Microservices auch Ereignisgetrieben erfolgen. Hierbei löst ein Microservice bei einem eingetretenen Ereignis ein Event aus und sendet dieses an einen Broker. Andere Microservices können sich bei diesem Broker anmelden und angeben, welche Events sie erhalten möchten. Das übergeordnete Microservice weiß dabei nicht, welche anderen untergeordneten Microservices diese Events erhalten. Dieses Vorgehen ermöglicht auch auf Kommunikationsebene eine Unabhängigkeit der Microservices.

Eine weitere Möglichkeit der asynchronen Kommunikation sind geteilte Datenspeicher zwischen Microservices. Gerade wenn bei der Kommunikation viele Daten ausgetauscht werden müssen, eignet sich zum Beispiel ein geteiltes Dateisystem, worauf mehrere Microservices zugreifen können.

2.3.2 Vorteile

Die vorgestellten Vorteile einer Microservice-Umgebung beziehen sich im Vergleich auf ein monolithisches System.

Durch die Unabhängigkeit der Microservices wird sich der Vorteile von Datenkapselung bedient. Dadurch kann die Implementierung eines Microservices beliebig geändert werden, solange die nach außen definierten Schnittstellen kompatibel bleiben. Die Datenkapselung ermöglicht zudem, die Technologien der Microservices frei zu wählen. So kann für die Umsetzung eines Microservices die passendsten Technologien wie zum Beispiel der Programmiersprache oder der Datenbank ausgewählt werden. Auch lassen sich Technologien leichter austauschen, da der Aufwand des Auswechselns einer Technologie für ein einzelnes Microservice geringer ausfällt.

Microservices werden typischerweise anhand von Geschäftsprozessen abgegrenzt. Hierdurch wird sich der Vorteile von Domain-Driven Design bereichert. Meist fallen Änderungswünsche oder Erweiterungen entlang von Geschäftsprozessen an. Da die Microservices ebenso an diesen Geschäftsprozessen entlang modelliert werden, reduziert sich der Aufwand von Änderungen oder Erweiterungen auf ein einzelnes Microservice.

Des Weiteren wird die Bereitstellung eines einzelnen Microservices erleichtert. Der Umfang eines einzelnen Microservice fällt in der Regel kleiner aus, wodurch sich die Komplexität als auch die Dauer der Bereitstellung verringert. Änderungen an einem Microservice können dadurch schneller ausgeliefert werden.

Microservices werden in einem verteilten System auf mehreren Servern betrieben. Daraus wird die Robustheit und Stabilität des Gesamtsystems gefördert, da das Gesamtsystem beim Ausfall eines einzelnen Servers weiterhin verfügbar bleibt. Auch die Skalierbarkeit der einzelnen Microservices verbessert die Robustheit. So können einzelne Microservice bei Bedarf entsprechend skaliert werden. Im Vergleich zu einem monolithischen Gesamtsystem kann beim horizontalen Skalieren, das heißt eine Software mehrfach auszuführen, Ressourcen und damit auch Kosten gespart werden, da gezielt nur die Komponenten skaliert werden können, auf welche gerade eine große Last liegt.

2.3.3 Herausforderungen

Eine Microservice-Architektur bringt eine höhere Komplexität des Gesamtsystems mit sich. Anstatt nur einen Monolithen zu verwalten, ist es bei Microservices nötig, den Überblick über alle Microservices und dessen Zusammenspiel zu behalten.

Da für jedes Microservice eigene Technologien verwendet werden können, kann es auch

zu einer Technologieüberlastung kommen. Dabei wird sehr viel Know-how über sämtliche Technologien benötigt.

Die lokale Entwicklung einer Microservice-Umgebung erschwert sich, da viele einzelne Microservices ausgeführt werden müssen, um das Gesamtsystem zu bilden. Wenn das Gesamtsystem eine gewisse Größe erreicht hat, kann es passieren, dass ein einzelner Rechner, auf welchem die Entwicklung stattfindet, nicht mehr alle Microservices ausführen kann, da die Rechenleistung einer einzelnen Maschine zu schwach ist. Dadurch müssen auf andere Mechanismen bei der Entwicklung zurückgegriffen werden. Wenn zum Beispiel ein bestimmtes Microservice entwickelt wird, dann können abhängige Microservices durch leichtgewichtige Simulationen ersetzt werden.

Ebenso erschwert sich die Überwachung und Fehlersuche von Microservices. Bei der Überwachung müssen alle Microservices betrachtet und bei einem auftretenden Fehler die Ursache in einem komplexeren System gefunden werden. Beispielsweise kann ein Fehler in der Benutzerschnittstelle nicht immer direkt einem bestimmten Microservice zugeordnet werden.

Die Sicherheit in einer Microservice-Architektur muss besonders bedacht werden. Da das Gesamtsystem in viele Microservices unterteilt wird, die jeweils eine eigene Schnittstelle nach außen anbietet, vergrößert sich die Angriffsfläche der Gesamtanwendung. Jedes Microservices muss hierbei separat vor Sicherheitslücken oder unbefugtem Zugriff geschützt werden.

Auch beim Testen stößt die Qualitätssicherung auf Hindernisse, da herkömmliche Teststrategien für Monolithen nicht mehr auf das Gesamtsystem angewendet werden können. Bei Microservices ist es erforderlich, neue Konzepte anzuwenden, um die korrekte Funktionsfähigkeit des Gesamtsystems zu validieren. Ebenso müssen die Interaktionen zwischen den Microservices gesondert betrachtet und getestet werden, damit die Integration der Microservices problemlos abläuft.

Da die Microservices in einzelnen isolierten Prozessen ausgeführt werden, erhöht sich die Latenz beim Aufruf anderer untergeordneter Microservices. Die Latenzunterschiede sind vor allem bei einer langen Kette von Aufrufen zu anderen Microservices spürbar. Ebenso kann durch die Teilung des Gesamtsystems die Datenkonsistenz zwischen verschiedenen Microservices nicht mehr garantiert werden.

3 Teststrategien für Microservice-Architekturen

In diesem Kapitel werden verschiedene Teststrategien vorgestellt, die für Microservice-Architekturen eingesetzt werden können. Abschnitt 3.1 erörtert die Notwendigkeit von automatisierten Tests. Der Abschnitt 3.2 präsentiert die Testpyramide, die vor allem in monolithischen Anwendung eingesetzt wird. Darüber hinaus wird in diesem Abschnitt analysiert, inwiefern die Testpyramide für Microservices angewendet werden kann. Abschnitt 3.3 erklärt Unit-Tests, die verwendet werden, um ein Microservice intern zu testen. In Abschnitt 3.4 werden Integrationstests vorgestellt, die den Einsatz abhängiger Dienste eines Microservices testen. Um ein Microservices als gesamte Einheit zu testen, werden in Abschnitt 3.5 die Komponententests präsentiert. Damit ein Microservice eine Schnittstelle zur Verfügung stellen und diese auch im Nachhinein ändern kann, ohne dass Nutzer des Microservices plötzlich Fehler erhalten, werden in Abschnitt 3.6 die Contract-Tests vorgestellt. Abschnitt 3.7 beschäftigt sich mit dem Testen der Benutzerschnittstellen. Zum Schluss werden in Abschnitt 3.8 End-To-End-Tests erklärt, die das gesamte System testen.

3.1 Testautomatisierung

Bevor eine Software in eine Produktivumgebung ausgeliefert wird, muss die Software getestet werden, um damit Fehler zu vermeiden und die Softwarequalität zu steigern. Heutzutage wird von den Softwareentwicklern eine zügige Implementierung von neuen Funktionen oder von Änderungswünsche erwartet, um am Markt mit Innovationen voranschreiten zu können. Diese schnelle Auslieferung soll dabei ohne Einschränkungen der Softwarequalität geschehen.

Traditionell wurde Software zunächst gebaut und in einer Testumgebung gestartet. In dieser Testumgebung konnten dann manuelle Tests durchgeführt werden, indem manuelle Eingaben an der Benutzeroberfläche vorgenommen wurden. Dabei wurde das System als Black-Box betrachtet. Manuelle Tests sind jedoch sehr zeitaufwendig, mühsam und langwierig. Auch sind sie sehr fehleranfällig, wenn die manuellen Tests nicht dokumentiert sind und nicht jedes Mal die gleichen Schritte durchgeführt werden. Zudem ist die Testabdeckung sehr gering, da nicht jeder Testfall aufgrund von zu hohem Aufwand ausgeführt

wird. Gerade bei großen Änderungen am Quellcode können vermehrt Fehler auftreten, die ohne ausführliche Tests nicht erkannt werden.

Die Testautomatisierung hilft dabei, manuelle Tests zu reduzieren und die Softwarequalität langfristig zu verbessern. Die Automatisierung erfolgt mithilfe einer Continuous-Integration Pipeline. Diese Pipeline kompiliert zuerst das Programm und führt danach automatisch verschiedene Tests aus. Die Pipeline kann auch als Continuous-Integration / Continuous-Delivery Pipeline, kurz CI/CD, erweitert werden. Dabei wird das kompilierte Artefakt auf eine Infrastruktur ausgeliefert. Auf der Infrastruktur können dann weitere Tests durchgeführt werden, welche die Software im laufenden Zustand prüfen.

Die automatisierten Tests bieten den Vorteil, dass diese bei jeder Änderungen am Quellcode ausgeführt werden und ein schnelles Feedback über die Testergebnisse an den Entwickler zurückgeben. So kann der Entwickler direkt fehlgeschlagene Tests inspizieren und Fehler frühzeitig beseitigen.

3.2 Testpyramide

Zunächst wird untersucht, wie sich klassische Teststrategien für Monolithen auf eine Microservice-Landschaft anwenden lassen. In der Praxis werden meist keine Microservice-Landschaften von Grund auf neu entwickelt, sondern bestehende Systeme und monolithische Anwendungen werden nach und nach aufgeteilt und in eine Microservice-Architektur integriert. Zu dieser Integration gehören auch bereits geschriebene Tests, welche in die neuen Microservices übernommen und übertragen werden.

In monolithischen Systemen wird häufig eine Testpyramide verwendet. Erstmalig stellte Cohn in seinem Buch „Succeeding with Agile“ die Testpyramide vor. Die Testpyramide ist, wie in Abb. 3.1 gezeigt, in die drei Ebenen Unit-Tests, Service-Tests und User-Interface-Tests, kurz UI-Tests, unterteilt. Die Form der Pyramide stellt dabei die Menge an Tests für die jeweiligen Ebenen dar.

Auf der untersten Ebene ordnen sich die Unit-Tests ein. Die Unit-Tests bilden das Fundament der Testpyramide und sollen demnach eine breite Testabdeckung aufweisen. Unit-Tests sind sehr schnell in ihrer Ausführung und können einfach geändert und gewartet werden. Da Unit-Tests sehr nahe an dem zu testenden Quellcode arbeiten, lassen sich diese sehr leicht automatisieren. Durch die Automatisierung und der schnellen Laufzeit von Unit-Tests liefern diese ein frühzeitiges Feedback über die Funktionstüchtigkeit des Programms. Zudem können Unit-Tests als Regressionstests eingesetzt werden. Dadurch werden auftretende Fehler bei neuen Änderungen am Quellcode direkt erkannt. Da die Unit-Tests einen atomaren Bestandteil testen und somit den Quellcode isoliert untersuchen, sind Ursachen für Fehler leichter auffindbar und oftmals direkt einer Zeile im

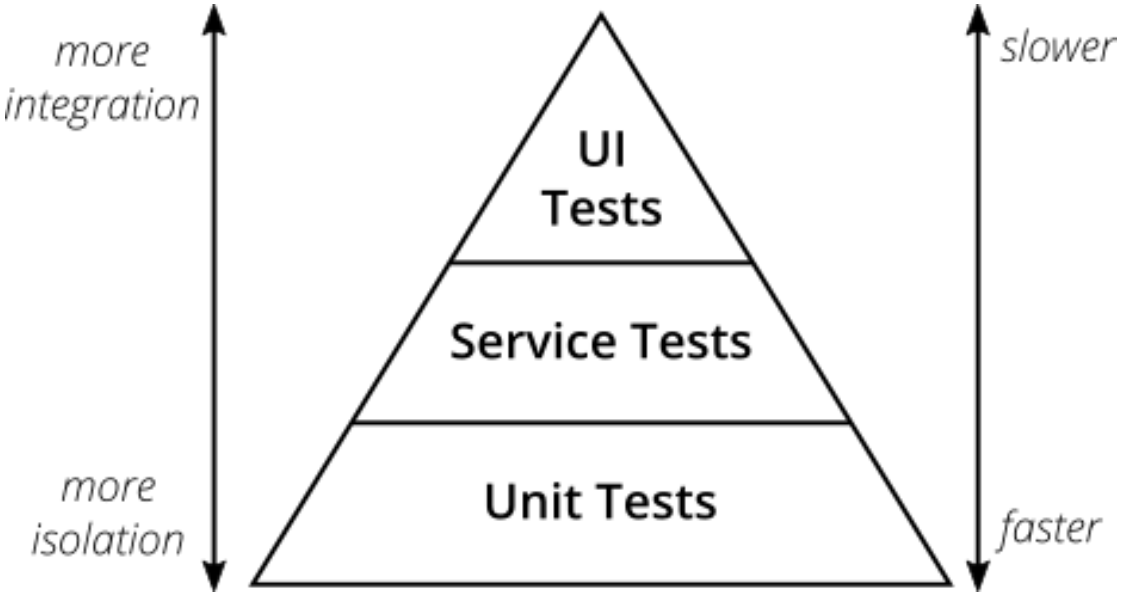

Abbildung 3.1: Testpyramide vorgestellt von Mike Cohn [2]

Quellcode zugeordnet.

In der mittleren Ebene der Testpyramide befinden sich die Service-Tests. Diese prüfen das Zusammenspiel verschiedener Komponenten in einer Anwendung. Mit Service-Tests lassen sich vor allem Schnittstellen zwischen den Komponenten des Systems kontrollieren. Dadurch steigt die Zuversicht über die Korrektheit einer Komponente und dessen Integration mit anderen Komponenten. Jedoch sind Service-Tests im Vergleich zu Unit-Tests aufwendiger zu implementieren und beanspruchen mehr Zeit in der Testdurchführung.

An der Spitze der Testpyramide sind die UI-Tests platziert. Diese testen die Benutzerschnittstellen und damit das Gesamtsystem als vollständige Anwendung. Erfolgreich ausgeführte UI-Tests bieten eine hohe Sicherheit, dass das Gesamtsystem fehlerfrei funktioniert. UI-Tests müssen sehr häufig angepasst und gewartet werden, weswegen sie nur in einer begrenzten Anzahl eingesetzt werden sollten. Da das gesamte System gestartet werden muss, sind UI-Tests auch deutlich langsamer im Vergleich zu Unit-Tests oder Service-Tests.

Die Pyramidenform visualisieren nicht nur die Menge an Tests für die jeweiligen Prüfebenen, sondern gibt auch einen Hinweis auf die benötigte Zeit für die Testdurchführung und welchen Grad an Isolation die Tests besitzen. Unit-Tests sind hierbei sehr schnell und testen den Quellcode in einer isolierten Umgebung. UI-Tests sind langsam in ihrer Durchführung und verlangen eine hohe Integration der Komponenten. Service-Tests halten sich in der Testdauer und der Isolation zwischen Unit-Tests und UI-Tests auf.

Die Testpyramide von Cohn wird heutzutage gerne, wie in Abb. 3.2 gezeigt, erweitert. So wird die Testpyramide mit manuellen Tests als Wolke über der Pyramide ergänzt. Manuelle Tests werden nur vereinzelt bei Bedarf eingesetzt und zum Beispiel bei Abnahmetests mit dem Kunden verwendet. Auch werden häufig die Service-Tests weiter in drei Ebenen unterteilt, um den Unterschied zwischen Komponententests, Integrationstests und API-Tests zu verdeutlichen [14]. Komponententests testen die Logik von spezifizierten Funktionalitäten, welche eine gesamte Komponente erfüllen soll. Integrationstests testen die Integration einer Komponente mit anderen Komponenten. API-Tests prüfen die Schnittstellen der Komponente nach Außen auf funktionale und nichtfunktionale Kriterien. Des Weiteren lassen sich auch End-To-End-Tests aus den UI-Tests abgrenzen [12]. End-To-End-Tests sind diejenigen Tests, die auf das Gesamtsystem angewendet werden. UI-Tests umfassen die End-To-End-Tests und inkludieren zudem Tests, welche die Benutzerschnittstellen isoliert prüfen.

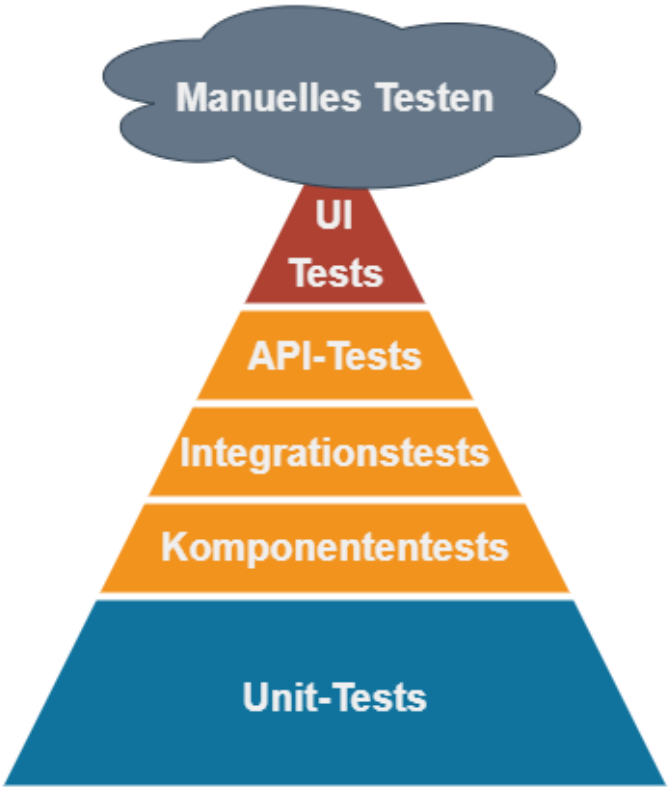

Abbildung 3.2: Erweiterte Testpyramide

Die Testpyramide wurde ursprünglich für monolithische Anwendungen konzipiert. Nun stellt sich die Frage, wie sich die Testpyramide auf eine Microservice-Architektur übertragen lässt und welche neuen Herausforderungen dabei entstehen. Ein einzelnes Microservice selbst lässt sich wie ein eigenständiger Monolith betrachten. Auf diese Weise kann

die Testpyramide jeweils direkt auf die einzelnen Microservices übertragen werden. Dieses Vorgehen beschreibt bereits Wolff in seinem Buch „Microservices: Grundlagen flexibler Softwarearchitekturen“ [15]. So betrachtet er zunächst das Gesamtsystem und wendet darauf die Testpyramide an. Auf der Ebene der Unit-Tests stehen die einzelnen Microservices und testen das Microservice völlig isoliert. Service-Tests prüfen dann das Zusammenwirken mehrerer Microservices. Mit UI-Tests wird schließlich die Benutzeroberfläche des Gesamtsystems auf Fehler untersucht. Neben der Anwendung der Testpyramide auf das Gesamtsystem wendet Wolff für jedes Microservices jeweils die Testpyramide erneut an. Hierbei überprüfen Unit-Tests kleinste Bausteine im Quellcode des Microservices. Verschiedene Komponenten innerhalb des Microservices werden anschließend mit Service-Tests integriert. UI-Tests eines Microservices betrachten die Schnittstellen des einzelnen Microservices, welche nach außen verfügbar gemacht werden.

3.3 Unit-Tests

Innerhalb eines Microservices gilt es zunächst, die einzelnen Bestandteile zu testen. Hierfür werden Unit-Tests eingesetzt, um die atomaren Bausteine des Programms zu untersuchen. Ein Unit-Test testet eine einzelne Funktion oder Methode im Quellcode, kann aber auch eine gesamte Klasse oder mehrere Klassen innerhalb eines Moduls behandeln. Unit-Tests überprüfen hauptsächlich die Logik des Programms und somit funktionale Anforderungen an das System. Dazu wird die zu testende Einheit mit Eingabedaten ausgeführt, die zuvor in einem Testfall definiert wurden. Die Ergebnisse dieser Operation werden anschließend mit den erwarteten Resultaten verglichen. Wenn die Ist-Ergebnisse nicht mit den Soll-Ergebnissen übereinstimmen, dann schlägt der Test fehl und informiert den Entwickler über einen Fehler.

Unit-Tests werden in der Regel mit derselben Technologie beziehungsweise derselben Programmiersprache entwickelt, in der das Programm implementiert wird. Dadurch ist es dem Entwickler möglich, zu seinem programmierten Quellcode die entsprechenden Unit-Tests umzusetzen. Wenn der Entwickler für seinen eigenen Quellcode die Tests entwirft, erfolgt die Umsetzung und Konstruktion meist als White-Box-Tests. Entscheidet sich der Entwickler dafür, die innere Struktur des Programmes in den Unit-Tests nicht zu berücksichtigen, dann handelt es sich um Gray-Box-Tests, da für ihn dennoch die Struktur bereits bekannt ist. Nur andere Entwickler, welche die internen Implementierungsdetails eines Programmteils nicht kennen, können die Unit-Tests als Black-Box-Tests umsetzen.

Mit Unit-Tests kann der gesamte Quellcode abgedeckt werden. In objektorientierten Sprachen empfiehlt es sich, für jede Klasse, die Geschäftslogiken enthält, eine Testklasse anzulegen, welche die öffentlichen Schnittstellen der Klasse testen. Zusätzlich können auch interne Funktionen einer Klasse getestet werden, jedoch erschweren Tests an internen Im-

plementierungen Veränderungen am Quellcode und können bei der Entwicklung stören. Deswegen sollten Tests interne Implementierungsdetails unberücksichtigt lassen. Ebenso sollten Tests für trivialen Quellcode vermieden werden. So ist es zum Beispiel nicht nötig, für Getter- und Setter-Methoden einer objektorientierten Klasse Unit-Tests zu schreiben.

Clemson teilt in seinem Artikel „Testing Strategies in a Microservice Architecture“ die Unit-Tests weiter in Sociable Unit-Tests und Solitary Unit-Tests ein. Geschäftslogiken werden mit Sociable Unit-Tests getestet und konzentrieren sich auf das Verhalten der einzelnen Komponenten innerhalb eines Microservices. Diese Tests betrachten eine Komponente als Black-Box und analysiert dessen Zustandsänderungen über die Schnittstellen der Komponente. Solitary Unit-Tests beschäftigen sich mit den Interaktionen zwischen einer Komponente und seinen Abhängigkeiten. Die Abhängigkeiten bei Solitary Unit-Tests werden simuliert, damit die zu untersuchende Komponente isoliert getestet werden kann. [16]

Nicht selten kommt es vor, dass eine Funktion eine Abhängigkeit innerhalb der Software benötigt, um seine Aufgabe zu erfüllen. Da Unit-Tests eine isolierte Einheit untersuchen, müssen Abhängigkeiten simuliert werden. Dadurch wird ausgeschlossen, dass eine Abhängigkeit der Grund für einen fehlschlagenden Test ist. Außerdem sollen Unit-Tests sehr schnell ausführbar sein, sodass zum Beispiel Datenbankzugriffe, Zugriffe auf das Dateisystem oder Abfragen über das Netzwerk nicht tatsächlich ausgeführt werden, da sie den Testlauf ansonsten stark verlangsamen würden. Um diese Abhängigkeiten zu simulieren, werden Test-Doubles eingesetzt. Es gibt die Möglichkeit, die Abhängigkeit mit einem Stub oder einem Mock zu ersetzen. Ein Stub ist eine primitive Implementierung der Abhängigkeit. Bei diesem gibt eine Funktion des Stubs einen statischen Wert zurück, der zuvor definiert wurde. Ein Mock hingegen enthält Logik und prüft zum Beispiel, ob eine Funktion der Abhängigkeit von der getesteten Funktion auch tatsächlich ausgeführt wird. [12]

Da Unit-Tests meistens mit derselben Technologie wie das Programm selbst implementiert werden, lassen sich Unit-Tests sehr gut automatisieren. Die Tests können oftmals bereits in der Entwicklungsumgebung ausgeführt werden oder als Teil des Kompiliervorgangs des Programms integriert werden. Dadurch erhält der Programmierer eine schnelle und frühzeitige Rückmeldung darüber, ob sein Quellcode Fehler verursacht. Des Weiteren lassen sich Unit-Tests in einer Continuous-Integration-Pipeline automatisieren, sodass neu eingepflegter Quellcode in einem Repository direkt auf Fehler untersucht werden. Somit werden Unit-Tests als Regressionstests eingesetzt, welche die neuste Version einer Software regelmäßig testet.

3.4 Integrationstests

Bei Unit-Tests werden externe Abhängigkeiten wie Datenbank- oder Dateizugriffe oder anderen Abfragen, die über das Netzwerk laufen, durch Imitate ersetzt. Der tatsächliche Zugriff auf diese externen Services wird jedoch nicht von Unit-Tests abgedeckt. Hierfür werden Integrationstests eingesetzt, welche die Interaktionen mit anderen Teilen des Systems testen. Gerade Stellen im Quellcode, wo Daten serialisiert oder deserialisiert werden, geben ein Indiz darüber, dass diese Stellen mit Integrationstests geprüft werden können.

Um die Integrationstests durchzuführen, wird das zu testende Microservice gestartet und ausgeführt. Ebenso werden die verwendeten externen Dienste, die den Integrationstest betreffen, lokal gestartet. Die externen Dienste können mit Test-Double ersetzt werden, um den Integrationstest von den externen Diensten zu entkoppeln und Fehlerquellen von äußeren Einflüssen zu vermeiden. Des Weiteren beschleunigen Test-Double die Testdurchführung, da diese lokal im Arbeitsspeicher ausgeführt werden können. Damit das Microservice im Integrationstest die lokal ausgeführten Dienste verwendet, muss das Programm einen Test-Modus unterstützen. In diesem Test-Modus können zum Beispiel Adressen für die verwendeten Dienste in der lokalen Umgebung eingestellt werden.

Clemson unterteilt Integrationstests weiter in Gateway-Integrationstests und Persistent-Integrationstests [16]. Gateway-Integrationstests prüfen die Netzwerkzugriffe auf Protokollebene auf Fehler. So werden unter anderem fehlende HTTP-Header oder ungültige SSL-Zertifikate geprüft. Ebenso stellen Gateway-Integrationstests sicher, dass das Microservice weiterhin fehlerfrei arbeitet, wenn externe Dienste fehlerhafte Antworten liefern oder diese gar komplett ausfallen. Persistent-Integrationstests untersuchen den verwendeten Datenspeicher. Dabei wird zum Beispiel das Schema der Daten mit dem im Datenspeicher eingesetzten Schema verglichen. Auch bei auftretenden Fehlern im Datenspeicher, testen Persistent-Integrationstests, ob das Microservice mit den Fehlern richtig umgeht.

3.5 Komponententests

Während Integrationstests Netzwerkzugriffe auf andere Dienste testen, beschäftigen sich Komponententests mit dem gesamten Microservice. Dabei kann das Microservice Komponententests für funktionale Kriterien unterzogen werden, um Geschäftslogiken zu überprüfen, aber auch für temporale Kriterien eingesetzt werden, um Laufzeitanforderungen oder Verhalten bei Überlast zu kontrollieren. Wie auch bei Integrationstests, werden externe Dienste und Abhängigkeiten durch Test-Double ersetzt, damit das Microservice isoliert betrachtet werden kann.

Bei Komponententests unterscheidet Clemson zwischen In-Process Komponententests und Out-Of-Process Komponententests [16]. Bei In-Process Komponententests werden Test-Double und Datenspeicher direkt im Arbeitsspeicher zusammen mit dem Microservice ausgeführt, um Netzwerkinteraktionen zu vermeiden. Um dies umsetzen zu können, benötigt das Microservice einen Test-Modus, damit die lokal im Arbeitsspeicher ausgeführten Dienste verwendet werden. In-Process Komponententests sind im Vergleich zu Out-Of-Process Komponententests schneller in der Testausführung. Out-Of-Process Komponententests setzen das unverändert Microservice ein, wodurch das Microservice für Out-Of-Process Komponententests keinen Test-Modus benötigt. Externe Dienste werden mit externen Test-Doublen ersetzt, auf welches das Microservice über das Netzwerk zugreift. So können in Out-Of-Process Komponententests zusätzlich Netzwerkkonfigurationen getestet werden. Im Vergleich zu In-Process Komponententests erhöht sich die Komplexität von Out-Of-Process Komponententests und durch die Netzwerkzugriffe verlangsamt sich zudem die Testdurchführung.

3.6 Contract-Tests

Gerade in Microservice-Architekturen konnten sich Contract-Tests zum Testen der API eines Microservices durchsetzen. Die Idee dahinter ist, dass ein Microservice, das ein anderes Microservice anspricht, in einem Vertrag angibt, welche Erwartungen es an die Schnittstelle des verwendeten Microservice stellt. Diese Erwartungen werden anschließend in Tests verpackt und an das andere Microservice weitergereicht. Das andere Microservice führt diese Tests dann aus und bestätigt damit, dass es die gewünschten Erwartungen erfüllt.

Ursprünglich wurden Microservices zunächst vollständig entwickelt und die API des Microservices in einer Schnittstellenspezifikation ausführlich dokumentiert. Danach konnten andere Microservices mithilfe dieser Dokumentation auf das Microservice zugreifen. Wenn jedoch Änderungen an der Schnittstelle zu einem späteren Zeitpunkt vorgenommen werden mussten, dann konnte nicht direkt festgestellt werden, ob eventuell Fehler bei Nutzern des Microservices auftreten. Dadurch war nie klar, welche Teile der Schnittstelle wie von anderen Microservices verwendet werden. Als Resultat konnten Schnittstellen kaum geändert und verbessert werden. Stattdessen wurden die Schnittstellen lediglich mit neuen Funktionen erweitert. Veraltete Funktionen verharrten in der API und wurden in der Dokumentation als veraltet gekennzeichnet, in der Hoffnung, dass Nutzer diese Funktionen mit den Neuen ersetzen. [17]

Contract-Tests gehen hingegen den Weg, dass die Nutzer vorgeben, wie sie die Schnittstelle eines Microservices einsetzen. Dabei gibt es zwei Rollen. Die erste Rolle ist die des sogenannten Consumers, der die Schnittstelle eines anderen Microservices verwendet. Die zweite Rolle ist der sogenannte Provider, der seine Schnittstelle für andere Microservices

anbietet. Ein Microservice kann dabei beide Rollen annehmen und sowohl als Consumer andere Microservices benutzen als auch als Provider eine eigene Schnittstelle für andere Microservices zur Verfügung stellen. Ein Provider kann mehrere Consumer besitzen, welche die API des Providers ansprechen. Jeder Consumer definiert zunächst seine eigene Consumer-Contracts und beschreibt darin, wie es das externe Dienstverhalten des Providers erwartet. Zu diesen Consumer-Contracts implementiert jeder Consumer seine entsprechenden Consumer-Contract-Tests. Anschließend werden alle Consumer-Contracts von den verschiedenen Consumern zusammengefasst, welche den Consumer-driven Contract ergeben. Aus den Consumer-driven Contracts ergeben sich die Consumer-driven Contract-Tests, worin alle Consumer-Contract-Tests jedes Consumers enthalten sind. Diese Consumer-driven Contract-Tests werden dann dem Provider übergeben, der diese Tests ausführt. [15]

Um die Übergabe der Tests an den Provider zu realisieren, können verschiedene Methoden angewendet werden. Eine einfache Möglichkeit ist das Ablegen der Tests im Repository des Consumers. Der Provider wird darauf aufmerksam gemacht, diese Tests aus dem Repository zu laden und auszuführen. Eine ähnliche Möglichkeit ist das Speichern der Tests im Repository des Providers. Dazu benötigt das Team, das den Microservice als Consumer implementiert, Zugriff auf das Repository des Providers, um die Contract-Tests dort einpflegen zu können. Eine Alternative, bei dem die Consumer und Provider unabhängig voneinander sind, ist das Hochladen der Tests bei einem dritten Dienst wie einem Artefakt Repository. Der Consumer setzt die Contract-Tests im eigenen Repository um. Anschließend lädt der Consumer die Contract-Tests an einem globalen Ort hoch und markiert dabei den entsprechenden Provider. Der Provider wiederum holt sich aus diesem globalen Ort alle für ihn markierten Tests, um diese dann auszuführen. Durch diesen Mechanismus können Consumer und Provider unabhängig voneinander agieren.

Der Provider führt alle Contract-Tests nach jeder Quellcodeänderung automatisiert im Rahmen der CI/CD-Pipeline aus, sodass nie inkompatible Versionen der Microservices in der Produktionsumgebung landen. Dadurch wird sichergestellt, dass die Zusammenarbeit alle Microservice einwandfrei läuft und es zu keinen Fehlern im System kommt.

Bei der Ausführung der Contract-Tests wird das vollständige Microservice auf seine API getestet. Abhängigkeiten des Microservices wie externe Dienste werden durch Simulationen ersetzt, um das Microservice in Isolation zu testen. Dadurch werden Fehler durch äußere Einflüsse wie Netzwerkkomplikationen vermieden. Zusätzlich werden die Contract-Tests auch gegen Stubs und Mocks des Microservices durchgeführt. So werden auch die Schnittstellen der Test-Doubles auf ihre Richtigkeit geprüft. Wenn der Provider diese Test-Doubles anbietet, werden alle Contract-Tests von allen Consumern ausgeführt. Falls ein Consumer eigene Stubs oder Mocks für das Microservice erstellt hat, wendet er nur seine selbst geschriebenen Contract-Tests auf die Test-Doubles an.

Der Provider dokumentiert zusätzlich in einem Provider-Contract seine Schnittstellenspezifikation und legt darin fest, welche Operationen verfügbar sind und wie diese richtig

verwendet werden. Darin werden unter anderem Datenformate und eingesetzte Übertragungstechnologien erläutert. Über diese Dokumentation werden andere Microservices bei der Benutzung der Schnittstelle unterstützt.

In den Contract-Tests werden ausschließlich Elemente der API getestet, auf welche auch vom Consumer zugegriffen wird. Wenn zum Beispiel in einer Antwort des Providers Daten ausgeliefert werden, die mehrere Attribute enthalten, der Consumer jedoch nicht alle Attribute davon ausliest, dann werden im Test entsprechend nur auf die benötigten Attribute eingegangen. Dadurch können Attribute, die von keinem Consumer ausgelesen werden, vom Provider gegebenenfalls entfernt oder verändert werden, ohne dass diese Änderung zu Fehlern im System führt. Wichtig für den Provider ist hierbei, dass Änderungen abwärtskompatibel für alle Consumer bleiben. [17]

Wenn Änderungen an der Schnittstelle vorgenommen werden, bei denen jedoch ein Contract-Test fehlschlägt, dann ist direkt bekannt, bei welchen anderen Microservice diese Änderung Auswirkungen hat. Dadurch ist es möglich, entsprechend auf den fehlschlagenden Tests zu reagieren. Entweder sorgt der Provider dafür, dass der Fehler in der eigenen Schnittstelle behoben wird, sodass die bestehenden Tests wieder erfolgreich durchlaufen, oder die Teams, welche die betroffenen Microservices entwickeln, treten in Kontakt und kommunizieren das weitere Vorgehen. So kann der Provider zum Beispiel Operationen als veraltet markieren und betroffene Microservices darüber informieren, dass diese Inkompatibilität in Zukunft aufgelöst werden muss. [12]

Da die Contract-Tests isoliert ausgeführt werden und keine externen Dienste benötigen, können die Tests sehr schnell durchlaufen werden und liefern zuverlässige Ergebnisse. Die Contract-Tests können sowohl für funktionale Anforderungen als auch für temporale Kriterien eingesetzt werden. Zum Beispiel kann ein Consumer einen Laufzeittest erstellen, welche die Zeitanforderungen an das Microservice überprüft.

3.7 UI-Tests

Um die Benutzerschnittstellen zu testen, werden User-Interface-Tests, kurz UI-Tests, eingesetzt. Zu den Benutzerschnittstellen eines Systems zählen nicht nur Benutzeroberflächen wie zum Beispiel eine Webseite, sondern auch Befehlszeilenschnittstellen über eine Konsole oder Schnittstellen, die über REST für Benutzer zugänglich sind.

UI-Tests stellen sicher, dass die Benutzerschnittstellen korrekt funktionieren. So testen diese unter anderem, ob bei einer Eingabe die richtigen Aktionen ausgelöst werden, Daten ordnungsgemäß dargestellt und präsentiert werden oder ob die Benutzerschnittstellen entsprechend ihren Status ändern. UI-Tests können beispielsweise das Verhalten einer Benutzerschnittstelle, das Layout oder die Benutzbarkeit auswerten, aber auch die Stim-

migkeit des Designs mit dem Unternehmensdesign vergleichen. [17]

Dazu können UI-Tests sowohl auf das Gesamtsystem angewendet werden, als auch ähnlich zu Unit-Tests implementiert werden. UI-Tests, die auf dem Gesamtsystem ausgeführt werden, gehören zur Kategorie der End-To-End-Tests, welche in Abschnitt 3.8 näher beschrieben werden. Beim Testen des Gesamtsystems ist die Zuversicht bei erfolgreichen Testergebnissen, dass alles gemeinsam richtig funktioniert, sehr hoch. Jedoch dauern diese Tests sehr lange bei der Ausführung, da alle Komponenten des Systems gestartet und betrieben werden müssen. Deswegen eignen sich unter anderem auch Unit-Tests für die Benutzeroberfläche, bei welchen die Systeme im Hintergrund durch Test-Double ersetzt werden. Zwar kann dabei nicht das Gesamtsystem auf die richtige Funktionsweise geprüft werden, jedoch können die einzelnen Elemente auf der Benutzeroberfläche getestet werden. So können zum Beispiel schon in Unit-Tests auf das Layout oder das Design eines Elements eingegangen werden.

Eine bewährte Technik, um UI-Tests umzusetzen, sind automatisierte Eingaben, die in der Benutzerschnittstelle eingegeben werden. Zur Validierung, dass die Benutzerschnittstelle korrekt reagiert hat, kann das Ergebnis mit dem erwarteten Ergebnis verglichen werden. Eine weitere Methode für die automatisierte Validierung sind erstellte Screenshots der Benutzeroberfläche, die mit älteren Versionen verglichen werden. Darin sucht ein Programm nach gravierenden Änderungen an der grafischen Darstellung und teilt diese dem Entwickler mit. Allerdings neigen UI-Tests häufig dazu, auch bei gewünschten Änderungen an der Oberfläche fehlzuschlagen, da zum Beispiel kleine Änderungen am Quellcode das Layout oder das Design verändern können. Deswegen müssen UI-Tests häufig gewartet und neu erstellt werden.

3.8 End-To-End-Tests

End-To-End-Tests testen das gesamte System mit der Zusammenarbeit aller Microservices. Das Ziel der End-To-End-Tests ist die Sicherstellung, dass das Gesamtsystem in der Produktivumgebung problemlos funktioniert. Daher werden End-To-End-Tests in einer Umgebung durchgeführt, welcher der Produktivumgebung sehr nahekommt. Dazu werden sowohl möglichst gleiche Softwarekomponenten als auch gleiche Hardware für die Infrastruktur verwendet, auf welcher die End-To-End-Tests ausgeführt werden. Daraus ergibt sich bei erfolgreich ausgeführten End-To-End-Tests eine große Zuversicht, dass das System am Schluss in der Produktion ordnungsgemäß arbeitet.

Für die Umsetzung der End-To-End-Tests wird das Gesamtsystem als Black-Box betrachtet. Customer Journeys, zu Deutsch auch Kundenreise genannt, können hergenommen werden, um einen Ablauf eines Kunden bei der Nutzung der Anwendung in End-To-End-Tests abzubilden [17]. Dadurch wird sichergestellt, dass ein Kunde sein Ziel mit

der Anwendung erreichen kann und es zu keinen Problemen kommt. Häufig werden diese End-To-End-Tests als Tests an der Benutzeroberfläche implementiert. End-To-End-Tests können aber auch anhand der äußeren Schnittstellen des Gesamtsystems getestet werden und so zum Beispiel über eine REST API das Gesamtsystem ansprechen.

Zwar geben End-To-End-Tests eine hohe Sicherheit über die korrekte Funktionalität des Gesamtsystems, aber sie sind sehr anfällig für unerwartete und unvorhersehbare Fehlern. Gerade gelegentliche Netzwerkfehler sind nicht untypisch bei einem verteilten System. Die Gefahr bei diesen Fehlern besteht darin, dass diese nicht konsistent auftreten. Das kann dazu führen, dass End-To-End-Tests nur manchmal erfolgreich durchlaufen, jedoch hin und wieder fehlschlagen. Diese Inkonsistenz der Testdurchführung kann die Entwickler dazu verleiten, die Testergebnisse zu ignorieren, mit der Annahme, dass die Fehlschläge nur temporär seien. Des Weiteren kann die Fehlerquelle eines fehlschlagenden End-To-End-Tests nicht direkt ermittelt werden. Es muss erst mühsam herausgearbeitet werden, welchem Microservice oder welcher Komponente im System der Fehler zuzuschreiben ist und wo sich der Fehler auch innerhalb eines Microservices befindet. End-To-End-Tests können nicht einem einzelnen Microservice zugeordnet werden. Da jedoch eine Microservice-Architektur darauf abzielt, die einzelnen Komponenten unabhängig voneinander zu entwickeln und regelmäßig neue Versionen der Microservices zu veröffentlichen, müssen End-To-End-Tests häufig gepflegt und gewartet werden, um auf die viele verschiedenen Änderungen zu reagieren. Bei dieser Pflege ist ebenso unklar, welches Team sich um die End-To-End-Tests kümmert und wie diese über Änderungen eines Microservices informiert werden. Zudem sind End-To-End-Tests sehr langsam in ihrer Durchführung. Um End-To-End-Tests ausführen zu können, müssen alle Microservices miteinander gestartet werden, was eine gewisse Zeit in Anspruch nimmt. Das Starten des gesamten Systems ist häufig sehr komplex und benötigt viele Ressourcen, sodass End-To-End-Tests nur in seltenen Fällen lokal auf einem Rechner durchgeführt werden können. [12]

Aufgrund der zahlreichen Hindernisse der End-To-End-Tests empfiehlt es sich, nur wenige dieser Tests einzusetzen. So sollten nur die wichtigsten Customer Journeys in End-To-End-Tests getestet werden, damit eine ausreichende Sicherheit über die Funktionalität gewährleistet wird.

4 Implementierung der Teststrategien

In diesem Kapitel werden die vorgestellten Teststrategien aus Kapitel 3 auf eine Beispielanwendung übertragen. Dazu wird zunächst die Beispielanwendung beschrieben, um anschließend konkrete Implementierungen für die verschiedenen Tests exemplarisch zu zeigen. Des Weiteren werden die erreichten Ziele sowie die Realisierbarkeit der Tests untersucht. Zusätzlich werden Vorteile und Hindernissen der Tests analysiert und die Teststrategien evaluiert.

4.1 Vorstellung der Beispielanwendung

Als Beispielanwendung wird ein vereinfachter Webshop verwendet. Mit dem Webshop können Kunden sich einen Katalog mit unterschiedliche Produkten anschauen, wie in Abb. 4.1 gezeigt wird. Über ein Schaltflächenelement kann ein beliebiges Produkt dem Warenkorb hinzugefügt werden.

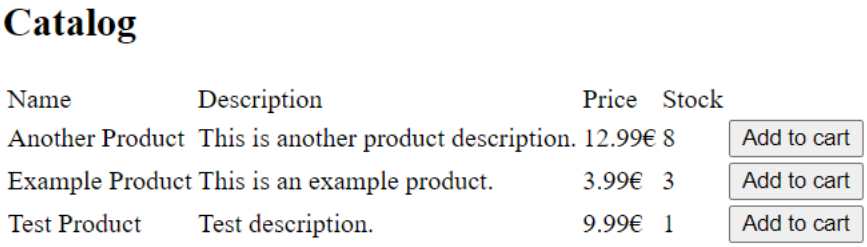

Abbildung 4.1: Katalogseite des Webshops

Der Warenkorb stellt schließlich, wie in Abb. 4.2 zu sehen, alle Produkte dar, die der Kunde dem Warenkorb hinzugefügt hat und gerne bestellen möchte.

Der Bestellprozess wird dann über ein entsprechendes Bedienelement fortgeführt, wodurch der Kunde bei der virtuellen Kasse landet. Der einfachheitshalber werden für diese Beispielanwendung keine Zahlungen absolviert, sondern lediglich die Kundeninformationen entgegengenommen. Abb. 4.3 zeigt das Eingabeformular für die Kundeninformationen. Über das Schaltflächenelement `Order` wird schlussendlich die Bestellung abgeschickt.

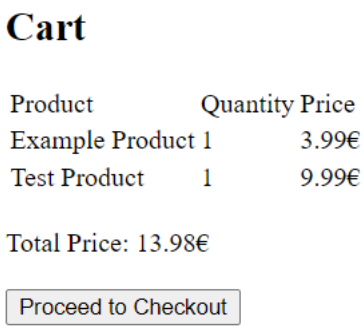

Abbildung 4.2: Warenkorbseite des Webshops

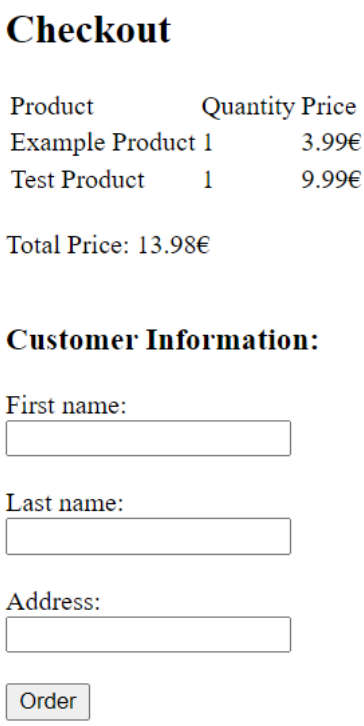

Abbildung 4.3: Kassenseite des Webshops

Auf der Bestellbestätigungsseite werden nochmals alle bestellten Produkte sowie die Kundeninformationen zusammengefasst, wie in Abb. 4.4 dargestellt.

Abbildung 4.4: Bestellbestätigungsseite des Webshops

Der Webshop ist dabei in fünf einzelne Microservices aufgeteilt, um daran die Teststrategien in einer Microservice-Architektur umzusetzen. Die Architektur des Webshops ist zur Übersicht in einem Komponentendiagramm dargestellt, das in Abb. 4.5 zu sehen ist. Die Microservices sind entlang von Geschäftsdomänen mithilfe von Domain-Driven Design festgelegt. Folgende Microservices bilden das Gesamtsystem des Webshops:

Catalog: Das Microservice Catalog kümmert sich um die angebotenen Produkte des Webshops. Im Katalog werden Namen, Beschreibungen und die jeweiligen Preise der Produkte gespeichert. Über eine Produkt-Id können mittels der Schnittstelle des Catalog-Microservices diese Produktdetails abgefragt werden. Zusätzlich können auch alle Produkte im Katalog erfragt werden, um eine Liste aller Produkte zu erhalten.

Stock: Mit der Lagerverwaltung der Produkte befasst sich das Microservice Stock. Hier werden zu den Produkten die aktuelle Anzahl sich im Lager befindlichen Artikel abgespeichert. Über die Schnittstelle des Microservices kann diese Anzahl abgefragt werden. Zudem kann die Menge der Artikel modifiziert werden, um so Artikel hinzuzufügen oder zu entfernen.

Customer: Um die Kundeninformationen zu verwalten, kommt das Microservice Customer zum Einsatz. Darin werden Vor- und Nachname sowie die Adresse des Kunden gespeichert. Über die Schnittstelle des Microservices können neue Kunden angelegt

werden und die Informationen über bereits existierende Kunden geladen werden.

Order: Die Ausführung der Bestellungen übernimmt das Microservice Order. Wenn ein Kunde eine neue Bestellung aufgibt, werden in diesem Microservice die Bestellinformationen abgelegt. Zu den Bestellinformationen gehört eine Kunden-Id, das Datum der Bestellung und die bestellten Produkte inklusive der Quantität und Preis für die Produkte. Das Order-Microservice überprüft beim Bestellprozess zunächst beim Catalog-Microservice, ob die angegebenen Produkte gültig sind und lässt dann die entsprechende Anzahl an Artikeln vom Stock-Microservice entfernen. Wenn bei der Bestellung ein Produkt ungültig ist oder nicht genügend Artikel im Lager sind, wird die Bestellung abgebrochen und ein Fehler ausgelöst.

Webshop-Client: Der Webshop-Client stellt die Benutzeroberfläche als Webseite für den Kunden bereit. Auf der Katalogseite werden alle Produkte vom Catalog-Microservice geladen und zusammen mit der Anzahl im Lager befindlichen Artikel, die über das Stock-Microservice abgefragt wird, angezeigt. Die einzelnen Produkte lassen sich dort dem Warenkorb hinzufügen. Der Webshop-Client merkt sich den Warenkorb anhand einer Session. Auf der Warenkorbseite werden nochmals alle Produkte angezeigt, die der Kunde gerne bestellen möchte, und der Gesamtpreis berechnet. Des Weiteren erfolgt auf der Kassenseite die Eingabe der Kundeninformationen, die bei Bestätigung der Bestellung an das Customer-Microservice weitergereicht werden. Außerdem leitet der Webshop-Client die Bestellung an das Order-Microservice weiter, um anschließend die Bestellbestätigung anzuzeigen.

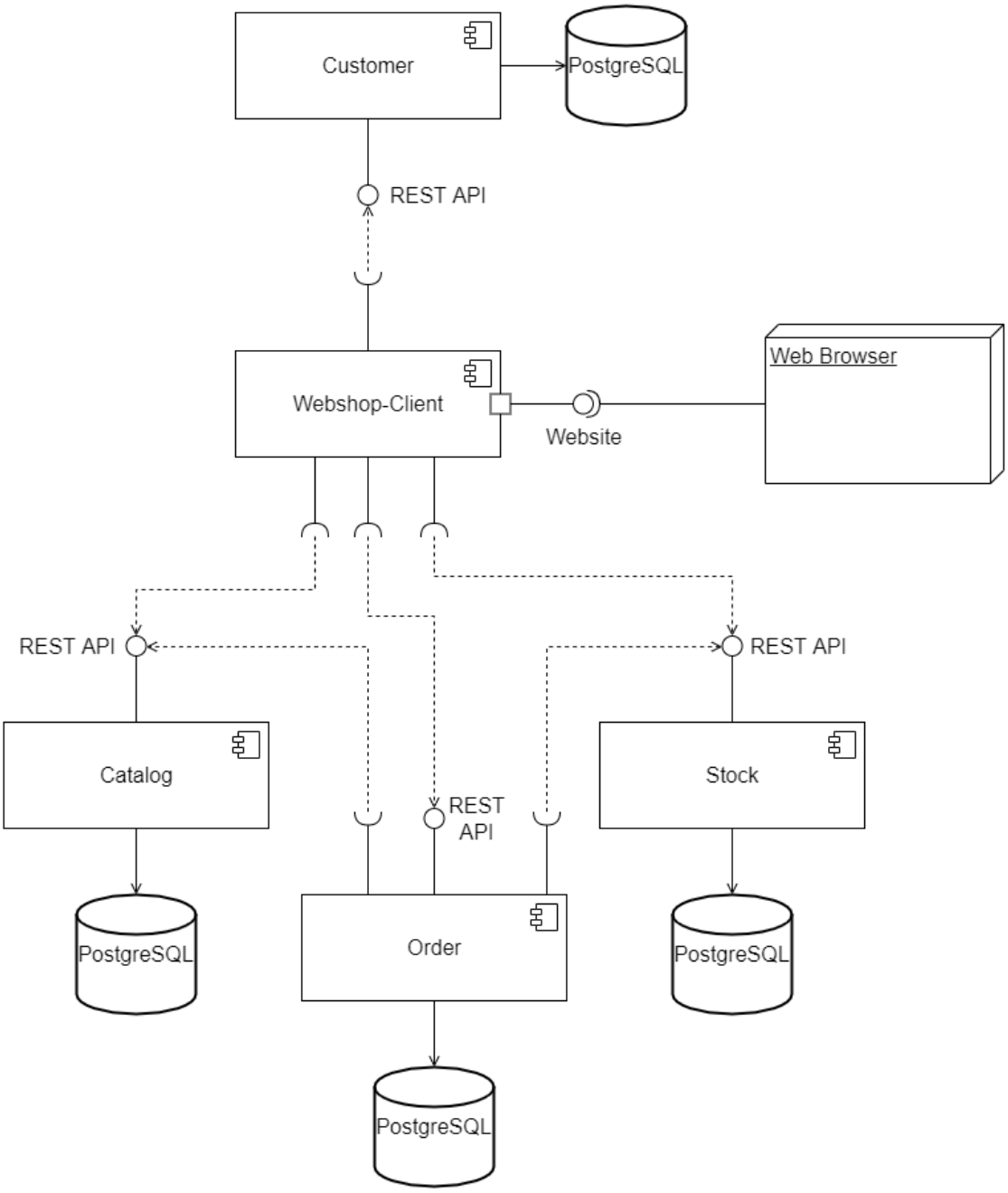

Abbildung 4.5: Komponentendiagramm des Webshops

Die Microservices sind alle in Java mit dem Spring Boot Framework entwickelt. Spring Boot baut auf das gleichnamige Spring Framework auf und erweitert dieses, um das Aufsetzen und Betreiben einer Spring-Anwendung zu erleichtern. Unter anderem integriert Spring Boot einen Webserver und legt Standardeinstellungen für die Anwendung fest. [18]

In den Microservices selbst wird intern eine Schichtenarchitektur verwendet. Eingesetzt werden drei Schichten: die Präsentationsschicht, die Logikschicht und die Datenhaltungsschicht. In der Präsentationsschicht werden REST-Controller eingesetzt, die für die Veröffentlichung der REST-Schnittstelle des Microservices verantwortlich sind. Eine Ausnahme macht hier das Webshop-Client-Microservice, welches stattdessen auf einen MVC-Controller zurückgreift und zusammen mit der Abhängigkeit Thymeleaf die Webseite des Webshops erzeugt. Thymeleaf ist eine serverseitige Template-Engine für Webanwendungen [19]. Die Präsentationsschicht greift auf die Logikschicht zu. In der Logikschicht kommen Service-Klassen zum Einsatz, welche die Geschäftslogiken der einzelnen Microservices enthalten. Daten kann die Logikschicht mithilfe der Datenhaltungsschicht ablegen und speichern.

Um den Datenspeicher umzusetzen, setzen die Microservices Spring Data JPA ein, das eine Erweiterung des Spring Boot Frameworks ist [20]. Die Abkürzung JPA steht für Java Persistence API und ist eine Spezifikation für Java [21]. Diese Spezifikation wird für Object-relational mapping, meist unter der Abkürzung ORM bekannt, genutzt. Die Idee davon ist, die Datenbanktabellen auf Objekte abzubilden und so eine Brücke zwischen der Datenbank und dem Programm zu schaffen. Mit dieser Abbildung wird die Entwicklung in einer objektorientierten Programmiersprache vereinfacht. JPA legt als Spezifikation nur den Rahmen fest, die konkrete Implementierung dieser Spezifikation erfolgt in Spring Data JPA mithilfe des Frameworks Hibernate. Als Datenbank verwenden die Microservices PostgreSQL, wobei jedes Microservice Zugriff auf eine eigene Datenbank erhält, damit die Microservices unabhängig voneinander laufen.

Damit die Microservices überhaupt miteinander kommunizieren können, müssen sie voneinander wissen, dass sie überhaupt existieren und wie sie sich gegenseitig erreichen können. Eine statische IP-Adresse für die Microservices kommt nicht infrage, da die Microservices zum einen lose gekoppelt sein sollen, zum anderen auch mehrere Instanzen eines Microservices betrieben werden, um die Anwendung auf Microservice-Ebene zu skalieren. Als Lösung kommt deswegen eine Diensterkennung zum Einsatz. Neu gestartete Microservices registrieren sich bei der Diensterkennung beziehungsweise bei der Service-Discovery. Wenn ein Microservice die Schnittstelle eines anderen Microservice ansprechen möchte, erkundet sich der Microservice bei der Diensterkennung nach Adressen des anderen Microservices, um so eine Anfrage an eine Instanz des anderen Microservice zu stellen. Eingesetzt wird in der Beispielanwendung Eureka, eine von Netflix bereitgestellte Open-Source-Software. Zusätzlich ermöglicht Eureka eine clientseitige Lastverteilung, wodurch Anfragen auf ein Microservices auf verschiedene Instanzen verteilt werden. Eureka wird mithilfe der Erweiterung Spring Cloud Netflix eingebunden [22].

4.2 Unit-Tests mit JUnit und Mockito

Zuerst werden Unit-Tests betrachtet. Dazu wird der in Listing 4.1 gezeigte Quellcode untersucht. Die Methode `getOrder` aus der Klasse `OrderService` ermittelt anhand einer Id eine Bestellung aus der Datenbank und gibt diese zurück. Wenn die Bestellung mit dieser Id nicht existiert, wird ein Fehler geworfen.

Listing 4.1: Methode aus der Klasse OrderService

```
1 public Order getOrder(UUID orderId) {
2     return orderRepository.findById(orderId)
3         .orElseThrow(() ->
4             new OrderNotFoundException(orderId));
5 }
```

Diese Methode ist in zwei Pfade unterteilt. Der erste Pfad resultiert aus einem existierenden Objekt aus dem `OrderRepository`, welches anschließend zurückgegeben wird. Der zweite Pfad entsteht aus einem fehlenden Objekt, wodurch ein Fehler erzeugt und geworfen wird. Da die Methode nicht allzu komplex ist, lässt sich diese Methode mit einer vollständigen Pfadabdeckung testen. Weil die Struktur der Methode zur Testabdeckung analysiert wurde, handelt es sich in den Tests um White-Box-Tests.

Die Unit-Tests sind ebenfalls wie die getestete Methode in Java geschrieben. Zum Testen wird das Framework JUnit 5 eingesetzt [23]. Um Abhängigkeiten innerhalb des Quellcodes zu simulieren, wird das Framework Mockito verwendet, welches sich sehr gut in JUnit integriert. Um Behauptungen aufzustellen, welche die Ist-Ergebnisse mit den Soll-Ergebnissen vergleichen, wird die Bibliothek AssertJ eingebunden.

Bevor Unit-Tests programmiert werden können, sind zunächst Vorbereitungen nötig. Um die Klasse `OrderService` zu testen, wird zuerst eine Klasse `OrderServiceTest` im Testmodul des Projekts erstellt. Diese Klasse ist in Listing 4.2 dargestellt. Gängigerweise wird für die Benennung der Testklasse derselbe Name wie die getestete Klasse verwendet und der Suffix `Test` angehängt. Die Klasse wird dabei in dasselbe Java-Paket gelegt. Der Hintergrund davon ist, dass so der Testklasse eine Paket-private Zugriffsberechtigung gewährt wird. In Listing 4.2 werden in der Methode `setUp` für die jeweiligen Tests Vorbereitungen getroffen. Diese Methode ist mit `@BeforeEach` markiert und teilt so JUnit mit, dass vor jedem Test diese Methode ausgeführt werden soll. In Zeile acht wird Mockito aufgefordert, alle mit `@Mock` markierten Attribute als Mocks zu initialisieren. Dadurch wird in Zeile vier ein Mock für die Abhängigkeit `OrderRepository` erzeugt. Daraufhin wird ein Objekt aus der zu untersuchenden Klasse `OrderService` erzeugt und diesem Objekt die simulierte Abhängigkeit übergeben.

Listing 4.2: OrderServiceTest-Klasse mit Vorbereitungen zum Testen der Klasse OrderService

```
class OrderServiceTest {
    private OrderService subject;
    @Mock
    private OrderRepository orderRepository;

    @BeforeEach
    void setUp() {
        openMocks(this);
        subject = new OrderService(orderRepository);
    }
}
```

Der erste Unit-Test aus Listing 4.3 testet, ob die Methode `getOrder` das Objekt `Order` entsprechend zurückgibt, wenn dieses Objekt gültig vom `OrderRepository` ausgegeben wird. Die Methode ist mit `@Test` markiert, damit JUnit diese Methode als Testmethode erkennt und ausführt. Üblicherweise wird eine Testmethode in drei Abschnitte unterteilt: Arrangieren, Handeln und Behaupten. Im ersten Abschnitt namens Arrangieren werden benötigte Variablen und Objekte angelegt. So wird in Zeile vier eine Id für die Bestellung angelegt und in Zeile fünf das erwartete Ergebnis erzeugt. In Zeile sieben und acht wird eine Simulation beziehungsweise ein Stub mit Mockito erstellt. Dieser Stub gibt vor, dass die Methode `findById` des Objekts `OrderRepository` das erwartete Objekt `Order` ausgibt. Die Erstellung des Stubs gehört ebenso noch zum Abschnitt des Arrangierens. Der zweite Abschnitt ist das Handeln, worin die zu testende Methode wie in Zeile elf ausgeführt wird. Schließlich werden im letzten Abschnitt Behauptungen formuliert. Entsprechend wird in Zeile 14 das Ist-Ergebnis mit dem Soll-Ergebnis verglichen. Ähnlich dazu wird im zweiten Unit-Test der Fall überprüft, dass das `OrderRepository` kein Ergebnis zurückliefert und deswegen die getestete Methode eine `OrderNotFoundException` werfen soll. Hier geschehen die Abschnitte des Handelns und Behauptens der einfachheitshalber zusammenhängend und sind nicht separat in einzelne Instruktionen aufgeteilt.

Listing 4.3: Testmethoden aus der Klasse OrderServiceTest zum Testen der Methode getOrder

```
@Test
void getOrder_Order_IfOrderExists() {
    // Arrange
    UUID orderId = UUID.randomUUID();
    Order expected = new Order(orderId, UUID.randomUUID(),
        Set.of(), System.currentTimeMillis());
    given(orderRepository.findById(orderId))
        .willReturn(Optional.of(expected));

    // Act
    Order actual = subject.getOrder(orderId);

    // Assert
    assertThat(actual).isEqualTo(expected);
```

```
15 }
16
17 @Test
18 void getOrder_ThrowsOrderNotFound_IfOrderIsUnknown() {
19     UUID orderId = UUID.randomUUID();
20     given(orderRepository.findById(orderId))
21         .willReturn(Optional.empty());
22     assertThatThrownBy(() -> subject.getOrder(orderId))
23         .isInstanceOf(OrderNotFoundException.class)
24         .hasFieldOrPropertyWithValue("orderId", orderId);
25     }
```

Durch die zwei Testmethoden wird eine vollständige Pfadabdeckung der Methode `getOrder` aus der Klasse `OrderService` erzielt, sodass die Logik der Methode getestet wird. Generell wird versucht, mit Unit-Tests eine hohe Testabdeckung nicht nur innerhalb einer Methode zu erzielen, sondern auch den gesamten Quellcode damit abzudecken. Dadurch wird sichergestellt, dass Geschäftslogiken richtig implementiert werden und verhindert, dass es zu Fehler in der Logik der Anwendung kommt. Jedoch ist es nicht bei allen Programmstücken möglich, diese vollständig mit allen Testfällen abzudecken. Gerade bei komplexen Methoden, die zum Beispiel eine verschachtelte Schleife enthalten, können nur wenige ausgewählte Testfälle in Unit-Tests umgesetzt werden, da ansonsten die Masse an Tests nicht mehr realisierbar wären. Des Weiteren sollte darauf geachtet werden, dass nicht alle Methoden einen separaten Unit-Test benötigen. So sollten private Methoden nicht getestet werden, da diese ein Implementierungsdetail der Klasse darstellen und sich häufiger ändern können. Ebenso sollten keine Methoden getestet werden, die keinerlei Verzweigungen enthalten und damit kaum eine Geschäftslogik abbilden. Der Aufwand der Implementierung und Wartung der Unit-Tests übersteigt bei diesen Methoden eher den Nutzen aus den Unit-Tests.

Ein großer Vorteil der Unit-Tests ist die schnelle Ausführung, wodurch der Entwickler direkt Rückmeldung über die Testergebnisse erhält. Auch ist die Fehlerursache bei einem fehlgeschlagenen Unit-Test direkt ersichtlich und einer konkreten Methode im Quellcode zuordenbar.

Die Unit-Tests in der Microservice-Architektur unterscheiden sich im Wesentlichen nicht von Unit-Tests einer monolithischen Architektur. Ein Unterschied besteht lediglich darin, dass die Tool-Auswahl in einem Monolith für den gesamten Monolith beschränkt ist. Microservices können hingegen mit unterschiedlichen Programmiersprachen und so auch unterschiedlichen Werkzeugen zum Testen entwickelt werden.

4.3 Persistent-Integrationstests mit JPA und Testcontainers

Als Nächstes werden Persistent-Integrationstests untersucht, die prüfen, dass ein Microservice ein Datenspeicher richtig einsetzt. Die Erweiterung Spring Data JPA ermöglicht die Erstellung eines Repositorys, worin eigene Objekte, die als Entität markiert sind, in der Datenbank gespeichert werden können. In Listing 4.4 ist das Repository für die Speicherung der Bestellungen abgebildet. Das Repository ist lediglich ein Interface mit zusätzlichen Methoden. Die konkrete Implementierung des Interfaces übernimmt das Framework. Die Methodennamen im Repository folgen einer bestimmten Vorgabe, damit das Framework die Semantik richtig aus dem Namen ableiten kann.

Listing 4.4: Interface OrderRepository mit Methode findOrdersByCustomerId

```
1 @Repository
2 public interface OrderRepository
3     extends JpaRepository<Order, UUID> {
4   List<Order> findOrdersByCustomerId(UUID customerId);
5 }
```

Auch bei den Integrationstests kommt wieder das Framework JUnit zum Einsatz. Zwar war dieses Framework ursprünglich für Unit-Tests gedacht, jedoch bietet JUnit ein ganzes Ökosystem für Tests an, welche neben Unit-Tests auch Einsatzzwecke für allerlei anderer Tests findet [23]. Spring Boot stellt in der Testerweiterung zusätzlich eine Testumgebung bereit, worin speziell Tests für JPA erstellt werden können. Dabei werden nur die relevanten Programmteile geladen, die für die Datenspeicherung relevant sind.

Da das Starten einer ganzen Datenbank in einem Testlauf einiges an Zeit beansprucht, wird die Datenbank mit einer In-Memory-Datenbank ausgetauscht. Als In-Memory-Datenbank wird H2 eingesetzt, die lokal im Arbeitsspeicher ausgeführt und so deutlich schneller beim Starten ist. Um jedoch auch die Konfiguration der Datenbank zu testen und eine ähnliche Anbindung zur Datenbank wie in der Produktivumgebung zu schaffen, werden Testcontainer verwendet, die mithilfe von Docker eine lokale PostgreSQL-Datenbank in einem Container startet.

In Listing 4.5 ist ein Persistent-Integrationstest dargestellt, der besonders die erstellte Methode im `OrderRepository` testet. Der Persistent-Integrationstest setzt auf die zuvor genannte In-Memory-Datenbank. In Zeile eins wird zuerst die Testklasse als JPA-Test markiert. Damit der Test nicht auf die Datenbank in der Produktivumgebung zugreift, sondern die In-Memory-Datenbank einsetzt, wird ein Test-Modus eingeschalten. Dazu wird in Zeile zwei dem Spring Boot Framework mitgeteilt, dass das Profil `in-memory-db` aktiviert werden soll. Bei der Aktivierung sucht das Framework nach einer Ressource namens `application-in-memory-db.properties`. In dieser Konfigurationsdatei wird schließlich der Datenbanktreiber auf H2 gestellt und die URL zum Datenspeicher entsprechend zu dieser Datenbank angepasst. Als Nächstes wird in Listing 4.5

ein Objekt der zu untersuchenden Klasse in Zeile fünf und sechs injiziert. Diese Injektion der Abhängigkeit übernimmt das Spring Framework. Die Methode `tearDown` stellt sicher, dass nach jedem einzelnen Test, der Datenspeicher geleert wird, sodass sich die Tests nicht untereinander beeinflussen. Ein konkreter Testfall wird dann in der Methode `findOrdersByCustomerId_OrderList_IfOrdersExist` geprüft, in welchem eine Bestellung als Rückgabewert erwartet wird, die zuvor gespeichert wurde. Der Ablauf des Persistent-Integrationstests ähnelt sich sehr dem der Unit-Tests. Zuerst werden im Abschnitt des Arrangierens alle für den Test benötigten Objekte erzeugt. In Zeile 19 wird eine Bestellung gespeichert, die anschließend in den Zeilen 20 bis 22 als Ausgabe von der zu untersuchenden Repository-Klasse erwartet wird. Im zweiten Test fällt der Abschnitt des Arrangierens weg, da das Repository ein leeres Ergebnis liefern soll, wenn zuvor keine Daten gespeichert wurden.

Listing 4.5: Testklasse für OrderRepository mit einer In-Memory-Datenbank

```
1  @DataJpaTest
2  @ActiveProfiles({"in-memory-db"})
3  class OrderRepositoryInMemoryTest {
4
5    @Autowired
6    private OrderRepository subject;
7
8    @AfterEach
9    void tearDown() {
10     subject.deleteAll();
11   }
12
13   @Test
14   void findOrdersByCustomerId_OrderList_IfOrdersExist() {
15     ProductOrder productOrder =
16         new ProductOrder(UUID.randomUUID(), 1, 1);
17     Order order = new Order(null, UUID.randomUUID(),
18             Set.of(productOrder), System.currenTimeMillis());
19     Order expectedOrder = subject.save(order);
20     assertThat(subject.findOrdersByCustomerId(
21             expectedOrder.getCustomerId()))
22         .contains(expectedOrder);
23   }
24
25   @Test
26   void findOrdersByCustomerId_Empty_IfOrderIsUnknown() {
27     assertThat(subject.findOrdersByCustomerId(
28             UUID.randomUUID()))
29         .isEmpty();
30   }
31 }
```

Der in Listing 4.5 gezeigte Quelltext verwendet eine In-Memory-Datenbank. Manche Da-

tenbanken unterscheiden sich jedoch von der Verhaltensweise zu der In-Memory-Datenbank. Um die Tests auf der Datenbank auszuführen, die auch in der Produktivumgebung verwendet wird, muss die entsprechende Datenbank lokal gestartet werden. In Listing 4.6 wird ein Persistent-Integrationstest gezeigt, der mithilfe von Testcontainer eine lokale PostgreSQL-Datenbank startet. Anstatt über einen Test-Modus eine In-Memory-Datenbank zu aktivieren, wird in Zeile zwei die Testklasse mit `Testcontainers` erweitert. In den Zeilen fünf bis zehn wird der PostgreSQL-Container angelegt und Parameter wie dem Benutzernamen übergeben. Die Erweiterung `Testcontainers` sorgt dafür, dass ein lokaler PostgreSQL-Container in Docker gestartet wird. In der folgenden Methode in Zeile 12 bis 21 werden Einstellungen an das Spring Framework übertragen, damit die Anwendung sich mit der lokal gestarteten Datenbank verbindet. Der Inhalt der Tests gleichen denen in Listing 4.5 bereits gezeigten Testmethoden.

Listing 4.6: Testklasse für OrderRepository mit einer externen Datenbank

```
1  @SpringBootTest
2  @Testcontainers
3  class OrderRepositoryExternalTest {
4
5    @Container
6    public static PostgreSQLContainer<?> container =
7        new PostgreSQLContainer<>(
8            "postgres:latest").withUsername("postgres")
9            .withUsername("postgres")
10           .withDatabaseName("postgres");
11
12   @DynamicPropertySource
13   static void properties(DynamicPropertyRegistry registry) {
14     registry.add("spring.datasource.url",
15         container::getJdbcUrl);
16     registry.add("spring.datasource.username",
17         container::getUsername);
18     registry.add("spring.datasource.password",
19         container::getPassword);
20   }
21 }
```

Die Persistent-Integrationstests stellen sicher, dass die Integration mit einem Datenspeicher problemlos funktioniert. Sie überprüfen, ob die Verbindung zur Datenbank erfolgreich aufgebaut werden kann und ob ein gültiges Schema vorliegt. Beim Testen des Schemas prüfen die Persistent-Integrationstests auch, ob die Entitäten richtig erstellt wurden, die ein Objekt auf eine Datenbanktabelle abbilden.

Bei der In-Memory-Datenbank wird ein Test-Modus benötigt, um die Datenbank auszutauschen. Zwar sind die Tests hierbei deutlich schneller in der Ausführung als wenn eine externe Datenbank gestartet werden muss, jedoch wird die Konfiguration für die in der Produktivumgebung verwendeten Datenbank nicht kontrolliert. Um diese Konfiguration

mit in den Test einzubeziehen, führt kein Weg daran vorbei, eine externe Datenbank für die Testdurchführung zu starten.

Persistent-Integrationstests können auch für monolithische Anwendung entwickelt werden. Ein Unterschied zur Microservice-Architektur besteht darin, dass jeder Microservice seine eigene Datenbank betreibt. Dabei können auch verschiedene Datenspeicher für die jeweiligen Einsatzzwecke der Microservices verwendet werden. Durch die verschiedenen Datenspeicher ist die Fehlerquelle bei der Integration im Vergleich zu einem Monolithen erhöht, wodurch die Persistent-Integrationstests in einer Microservice-Architektur eine wichtigere Rolle spielen.

4.4 Gateway-Integrationstests mit WireMock

Ähnlich zu den Persistent-Integrationstests muss auch die Integration mit externen Diensten getestet werden. Dazu werden Gateway-Integrationstests verwendet. In Listing 4.7 ist ein Quellcode aus dem Microservice Order angegeben, der auf das Microservice Catalog zugreift, um darüber den Preis eines Produktes zu ermitteln. Die Ermittlung erfolgt über die REST-Schnittstelle des Microservices Catalog. Um eine Anfrage an die Schnittstelle zu erzeugen, wird ein `WebClient` eingesetzt, der Teil der Bibliothek WebFlux ist [24]. WebFlux ermöglicht die Umsetzung des reaktiven Programmierparadigmas, wodurch die Anfrage asynchron ausgesendet wird. In Listing 4.7 wird der `WebClient` aufgerufen, um eine Get-Methode an den Pfad `/products/{productId}` zu senden. Das Ergebnis wird im JSON-Format erwartet, damit es in Zeile sieben zu einem Objekt konvertiert werden kann. Wenn ein Fehler bei der Anfrage auftritt, wird eine `ProductNotFoundException` geworfen.

Listing 4.7: Methode aus der Klasse CatalogService des Order-Microservices

```
1  public Mono<ProductDto> getProduct(UUID productId) {
2      return webClient.get()
3          .uri(uriBuilder -> uriBuilder
4              .path("/products/{productId}")
5              .build(productId))
6          .retrieve()
7          .bodyToMono(ProductDto.class)
8          .onErrorMap(error ->
9              new ProductNotFoundException(productId));
10 }
```

Um den externen Dienst zu simulieren, wird auf WireMock zurückgegriffen. WireMock startet einen lokalen Webserver, worauf die simulierten Ressourcen und deren Antworten laufen. Im Beispiel von Listing 4.8 wird zuerst das Microservice in einer Testumgebung gestartet. Dazu wird in Zeile eins die Testklasse als `SpringBootTest` markiert und in Zeile

zwei der Test-Modus aktiviert. Wie bereits in Abschnitt 4.3 beschrieben, werden dadurch Einstellungen für die Integrationstests vorgenommen. Genauer genommen wird die URL des Catalog-Microservices angepasst, damit diese auf den WireMock-Server zeigt. In Zeile drei wird schließlich WireMock angewiesen einen Webserver auf Port `23456` zu betreiben. In Zeile sechs und sieben wird die zu untersuchende Klasse mittels Spring injiziert. In der nachfolgenden Testmethode in Listing 4.8 wird getestet, ob ein Produkt richtig vom externen Dienst empfangen und verarbeitet wird. Zuerst werden nötige Variablen und Objekte angelegt. In den Zeilen 16-20 wird ein Stub erzeugt, der dafür sorgt, dass WireMock bei einer Get-Anfrage auf den Pfad `/products/{productId}` ein Produkt im JSON-Format ausgibt. Da die Anfrage asynchron mithilfe von WebFlux gestellt wird, kommt ein `StepVerifier` aus der Bibliothek `reactor-test` zum Einsatz, der ähnlich zu AssertJ Behauptungen für die Tests aufstellt. So wird in Zeile 21 ein StepVerifier mit dem asynchronen Ergebnis aus der zu testenden Klasse erstellt. Darauffolgend wird geprüft, ob das erwartete Produkt nach der asynchronen Anfrage zurückgegeben wird und keine Fehler geworfen wurden. Abgeschlossen wird der `StepVerifier` mit der Methode `verify()`.

Listing 4.8: Integrationstest im Order-Microservice für die externe Abhängigkeit zum Catalog-Microservice

```
1  @SpringBootTest
2  @ActiveProfiles("wiremock")
3  @WireMockTest(httpPort = 23456)
4  class CatalogServiceIntegrationTest {
5
6    @Autowired
7    private CatalogService subject;
8
9    @Test
10   void getProduct_Product_IfProductExists() throws JSONException {
11     UUID productId = UUID.randomUUID();
12     ProductDto productDto = new ProductDto(productId, 1.0);
13     JSONObject productJsonResponse = new JSONObject();
14     productJsonResponse.put("id", productId.toString());
15     productJsonResponse.put("price", productDto.getPrice());
16     stubFor(get("/products/" + productId)
17         .willReturn(ok()
18             .withHeader(HttpHeaders.CONTENT_TYPE,
19                 MediaType.APPLICATION_JSON_VALUE)
20             .withBody(productJsonResponse.toString())));
21     StepVerifier.create(subject.getProduct(productId))
22         .expectNext(productDto)
23         .expectComplete()
24         .verify();
25   }
26 }
```

Ähnlich zu den Persistent-Integrationstests kontrollieren die Gateway-Integrationstests die Kommunikation mit externen Diensten. Dabei können die Tests auch auf Protokollebene Untersuchungen durchführen und zum Beispiel den Header der HTTP-Anfrage auswerten. So wird sichergestellt, dass das Microservice korrekte Anfragen an externe Dienste stellt, die Daten richtig empfängt, deserialisiert und weiterverarbeitet.

Ein Problem bei den Gateway-Integrationstests entsteht dann, wenn sich die Schnittstelle des externen Dienstes ändert. Der externe Dienst kann bei Änderungen nicht erkennen, welche Anwendungen darauf zugreifen und wo eventuelle Fehler auftreten werden. Wenn der externe Dienst selbst entwickelt wird, eignen sich die in Abschnitt 3.6 beschriebenen Contract-Tests besser. Gateway-Integrationstests sind dann interessant, wenn der externe Dienst nicht selbst entwickelt und so kein Einfluss auf diesen genommen werden kann. Zum Beispiel, wenn in der eigenen Anwendung auf ein Bezahlsystem von einem externen Anbieter zurückgegriffen wird, stellen Gateway-Integrationstests die erfolgreiche Integration des externen Dienstes in die eigene Anwendung sicher. Da aber weiterhin der externe Dienst unvorhersehbare Änderungen an der Schnittstelle vornehmen kann, empfiehlt es sich, die Gateway-Integrationstests regelmäßig auszuführen. Dadurch können Fehler bei der Integration früh erkannt und gezielt darauf reagiert werden.

Gateway-Integrationstests können in gleicher Form auch in einer monolithischen Anwendung implementiert werden, um die Integration von externen Diensten zu testen. Während in einem Monolithen in der Regel nur an einer Stelle im Programm die Integration eines externen Dienstes erfolgt, erfordert eine Microservice-Architektur, dass diese Tests für jeden Microservice separat entwickelt werden. Dadurch erhöht sich die Komplexität und der Aufwand für die Wartung der Gateway-Integrationstests bei einer Microservice-Architektur.

4.5 Komponententests mit WebFlux

Um ein Microservice im Ganzen zu testen, werden Komponententests eingesetzt. Zuerst wird die Schnittstelle des Microservices betrachtet, damit anschließend dafür Tests entwickelt werden können. In Listing 4.9 wird diese Schnittstelle definiert. Unter der Ressource `/orders/{orderId}` können per Get-Methode Informationen einer Bestellung abgerufen werden. Eine neue Bestellung kann mit der Post-Methode auf der Ressource `/orders` erstellt werden.

Listing 4.9: OrderController definiert REST-Schnittstelle des Order-Microservices

```
1 @RestController
2 @RequestMapping("/orders")
3 public class OrderController {
4
5   private final OrderService orderService;
```

```
6
7     public OrderController(OrderService orderService) {
8       this.orderService = orderService;
9     }
10
11    @GetMapping("/{orderId}")
12    public Order getOrder(@PathVariable(name = "orderId") UUID
          orderId) {
13      return orderService.getOrder(orderId);
14    }
15
16    @PostMapping
17    public Mono<Order> createOrder(@RequestBody OrderDto orderDto) {
18      return orderService.createOrder(orderDto);
19    }
20  }
```

Damit Anfragen an das Microservice gestellt werden können, benötigen die Komponententests einen Klienten zur Erstellung von REST-Anfragen. WebFlux bietet dazu speziell einen `WebTestClient` an. Externe Abhängigkeiten werden in Komponententests durch Test-Double ersetzt. Hierfür wird wieder auf die in Abschnitt 4.3 bereits verwendete In-Memory-Datenbank H2 zurückgegriffen. Zur Simulation anderer Microservices wird WireMock eingesetzt, das in Abschnitt 4.4 vorgestellt wurde.

Die Komponententests für das Order-Microservice können aus Listing 4.10 ausgelesen werden. Dort wird in Zeile eins ein Spring Boot Test deklariert, wodurch das Microservice auf einem zufälligen Port ausgeführt wird. In der zweiten Zeile werden Einstellungen für das Spring Framework vorgenommen, um die Datenbank mit der zuvor vorgestellten In-Memory-Datenbank H2 auszutauschen und um die Adressen der abhängigen Microservices für WireMock anzupassen. Wie in den Gateway-Integrationstests wird in Zeile drei WireMock angesprochen, um einen lokalen Webserver mit den Test-Doublen zu betreiben. Da der Port des zu testenden Microservice benötigt wird, wird in Zeile acht und neun der Port geladen. Die Zuweisung des Wertes für den Port übernimmt das Spring Framework. In der folgenden Methode `setUp()` wird der `WebTestClient` erzeugt, welcher zuständig ist, Anfragen an das zu testende Microservice zu stellen und zu validieren. Wie bereits in den Persistent-Integrationstests erläutert, wird in der nachfolgenden Methode `tearDown()` die Datenbank geleert, damit sich die verschiedenen Tests nicht gegenseitig beeinflussen. Ein konkreter Testfall wird in den Zeilen 26 bis 47 durchgeführt. In diesem Testfall wird überprüft, ob eine Bestellung vom Microservice ausgegeben wird, wenn die Bestellung existiert. Darin wird im ersten Abschnitt der Testfall vorbereitet und die Bestellung in der Datenbank abgespeichert. Im zweiten Teil wird in den Zeilen 33 bis 37 der `WebTestClient` dazu aufgerufen, eine entsprechende Abfrage für die Bestellung zu erstellen. Anschließend werden in den Zeilen 38 bis 46 die vom Microservice zurückgelieferte Antwort im JSON-Format ausgewertet. Dazu werden alle Daten mit dem erwarteten Inhalt verglichen. So wird zum Beispiel in den Zeilen 41 und 42 die Id der Bestellung auf

ihre Richtigkeit geprüft.

In der vorgestellten Testmethode in Listing 4.10 wurde zur übersichtlicheren Darstellung kein Stub mithilfe von WireMock angelegt, da das Microservice bei der Ausgabe einer Bestellung keinen Zugriff auf abhängige Microservices vornimmt. Bei einer Testmethode, welche das Anlegen einer neuen Bestellung überprüft, ist es erforderlich, Stubs für das Catalog- und Stock-Microservice zu erzeugen, da diese bei diesem Prozess aufgerufen werden.

Listing 4.10: Komponententests für das Order-Microservice

```
1  @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
2  @ActiveProfiles({"in-memory-db", "wiremock"})
3  @WireMockTest(httpPort = 23456)
4  class OrderComponentTest {
5
6    @Autowired
7    private OrderRepository orderRepository;
8    @LocalServerPort
9    private int port;
10   private WebTestClient webTestClient;
11
12   @BeforeEach
13   void setUp() {
14     webTestClient = WebTestClient.bindToServer()
15         .baseUrl("http://localhost:" + port)
16         .defaultHeader(HttpHeaders.CONTENT_TYPE,
17             MediaType.APPLICATION_JSON_VALUE)
18         .build();
19   }
20
21   @AfterEach
22   void tearDown() {
23     orderRepository.deleteAll();
24   }
25
26   @Test
27   void getOrder_Order_IfOrderExists() {
28     ProductOrder productOrder =
29         new ProductOrder(UUID.randomUUID(), 1, 1.0);
30     Order order = new Order(null, UUID.randomUUID(),
31         Set.of(productOrder), System.currentTimeMillis());
32     orderRepository.save(order);
33     webTestClient.get()
34         .uri(uriBuilder ->
35             uriBuilder.path("/orders/{orderId}")
36                 .build(order.getId()))
37         .exchange()
38         .expectStatus()
```

```
39          .isOk()
40          .expectBody()
41          .jsonPath("$.id")
42          .isEqualTo(order.getId().toString())
43          .jsonPath("$.customerId")
44          .isEqualTo(order.getCustomerId().toString())
45          .jsonPath("$.orderTimestamp")
46          .isEqualTo(order.getOrderTimestamp())
47          .jsonPath("$.productOrders[0].productId")
48          .isEqualTo(productOrder.getProductId().toString())
49          .jsonPath("$.productOrders[0].quantity")
50          .isEqualTo(1)
51          .jsonPath("$.productOrders[0].price")
52          .isEqualTo(1.0);
53    }
54  }
```

Die Komponententests verfolgen das Ziel, das gesamte Microservice als eine Einheit zu testen. Dadurch wird sichergestellt, dass die einzelnen Elemente innerhalb eines Microservices korrekt miteinander interagieren und das Microservice richtige Antworten liefert. Zu einer richtigen Antwort gehört zum einen der Inhalt der Daten, aber auch das Format oder weitere Parameter auf Protokollebene, die genauer untersucht werden können.

Komponententests werden generell in In-Process und Out-Of-Process Komponententests unterteilt. Diese Unterteilung erfolgt jedoch nicht strikt, wie der Test aus Listing 4.10 zeigt. Im gezeigten Beispiel wird als Datenbank eine In-Memory-Datenbank eingesetzt, welche der Idee eines In-Process Komponententest folgt. Zur Simulation externer Abhängigkeiten wird hingegen ein lokaler Webserver gestartet, welcher sich in die Out-Of-Process Komponententests einordnet. Demnach sind auch Mischformen von In-Process und Out-Of-Process Komponententests praktikabel.

Die Komponententests bieten den Vorteil einer hohen Zuversicht darüber, dass das gesamte Microservice ordnungsgemäß arbeitet. Dazu können sowohl Geschäftslogiken überprüft werden, als auch operationale Aspekte berücksichtigt werden. Je nach Aufsetzen eines Komponententests verlängert sich gerade bei Out-Of-Process Komponententests die Testdurchführung. Hierdurch beanspruchen die Komponententests mehr Zeit bei ihrer Ausführung, wodurch der Entwickler eine verzögerte Rückmeldung über die Testergebnisse der Komponententests erhält.

Für monolithische Anwendungen sind Komponententests eher untypisch, da vergleichbare Tests bei Monolithen, welche die gesamte Anwendung testen, in die Kategorie der End-To-End-Tests fallen. In einer Microservice-Architektur hingegen prüfen die Komponententests isolierte Microservices ohne Rücksichtnahme auf andere beteiligte Microservices.

4.6 Contract-Tests mit Pact

In einer Microservice-Architektur konnten sich Contract-Tests zur Überprüfung der Kommunikation zwischen den Microservices bewähren. Contract-Tests werden von Nutzern eines Microservices erstellt und anschließend dem Microservices zur Ausführung bereitgestellt. Dadurch werden Inkompatibilitäten bei anderen Microservices bei einer Änderung an der Schnittstelle direkt detektiert.

Als Beispiel wird die Kommunikation zwischen den Microservices Order und Catalog untersucht. Das Microservice Order ist hierbei der Nutzer und ermittelt den Preis für ein Produkt, wie in Listing 4.7 bereits gezeigt wurde. Das Catalog-Microservice ist der Provider und bietet die REST-Schnittstelle an, worüber Details über die Produkte angefragt werden können.

Um die Contract-Tests zu erstellen, wird das Werkzeug Pact eingesetzt. Im ersten Schritt wird die Erstellung der Consumer-Contract-Tests angeschaut, welche im zweiten Schritt dann vom Provider ausgeführt werden.

In Listing 4.11 ist ein Consumer-Contract-Test dargestellt, der kontrolliert, dass ein existierendes Produkt richtig geladen wird. Zuallererst wird in Zeile eins die Testklasse als Spring Boot Test gekennzeichnet, wodurch das Microservice in einer Testumgebung gestartet wird. Passend dazu wird in Zeile zwei ein Test-Modus aktiviert, um die URL des abhängigen Microservices zu setzen. Die Integration von Pact in der Testdurchführung geschieht in den Zeilen drei bis fünf. Dort wird die Testklasse für Pact als Consumer-Test markiert und angegeben, für welchen Provider die Tests erstellt werden. Zusätzlich startet Pact einen lokalen Webserver und stellt einen Mock bereit, um die Consumer-Tests darauf ausführen zu können [25]. Dadurch wird sichergestellt, dass die Consumer-Contract-Tests ordnungsgemäß umgesetzt wurden, sodass die Integration aus der Seite des Consumers fehlerfrei läuft. In Zeile neun wird ein Objekt `CatalogService` mittels Spring injiziert, das später bei der Testausführung benötigt wird. In Zeile elf wird eine Produkt-Id erzeugt, die anschließend für die Erstellung des Vertrags und dem dazugehörigen Test benötigt wird. In den Zeilen von 13 bis 31 wird schließlich in der Methode `pactProductExists` ein Consumer-Contract definiert. Dabei wird im Vertrag zuerst in Zeile 19 der Zustand `product exists` zusammen mit der Produkt-Id angegeben, in welchem sich das Microservice bei der Ausführung der Tests befinden soll. Danach folgt eine Beschreibung des Vertrags. Wichtig für den Vertrag sind weiter die Angaben des Pfades und der HTTP-Methode, welche die Anfrage des Order-Microservice an das Catalog-Microservice abbilden. Anschließend wird die erwartete Antwort definiert, wobei in Zeile 25 bis 27 der erwartete Header, in Zeile 28 der erwartete Statuscode und in Zeile 29 der erwartete Inhalt der Antwort angegeben wird. Bei der Angabe des erwarteten Inhalts, wie in den Zeilen 16 bis 18 zu sehen, werden nur die Attribute mit einbezogen, die der Order-Microservice tatsächlich benötigt. Nachdem der Consumer-Contract-Test erstellt wurde, wird dieser noch gegen das Order-Microservice getestet. Dazu wird in Zeile 34 auf

den zuvor erstellen Vertrag verwiesen. In der Testmethode wird zunächst das erwartete Objekt erzeugt und anschließend mithilfe von `StepVerifier` festgestellt, ob der Mock, welcher Pact aus dem Vertrag generiert, richtig funktioniert und das erwartete Objekt zurückgibt.

Listing 4.11: Consumer-Contract-Tests des Microservices Order für den Provider Catalog

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@ActiveProfiles({"pact", "in-memory-db"})
@ExtendWith(PactConsumerTestExt.class)
@PactTestFor(providerName = "catalog", port = "23456",
    pactVersion = PactSpecVersion.V3)
class CatalogConsumerTest {

  @Autowired
  private CatalogService catalogService;

  private final UUID productIdExists = UUID.randomUUID();

  @Pact(consumer = "order")
  public RequestResponsePact pactProductExists(
      PactDslWithProvider builder) {
    DslPart responseBody = new PactDslJsonBody()
        .stringValue("id", productIdExists.toString())
        .decimalType("price", 1.0);
    return builder.given("product exists",
            "productId", productIdExists.toString())
        .uponReceiving("getProduct by productId")
        .path("/products/" + productIdExists)
        .method("GET")
        .willRespondWith()
        .headers(
            Map.of(HttpHeaders.CONTENT_TYPE,
                MediaType.APPLICATION_JSON_VALUE))
        .status(200)
        .body(responseBody)
        .toPact();
  }

  @Test
  @PactTestFor(pactMethod = "pactProductExists")
  void getProduct_Product_IfProductExists() {
    ProductDto productDto =
        new ProductDto(productIdExists, 1);
    StepVerifier.create(
            catalogService.getProduct(productIdExists))
        .expectNext(productDto)
        .expectComplete()
        .verify();
  }
```

```
44 }
```

Beim Ausführen der Consumer-Contract-Tests werden Pact-Dateien im JSON-Format generiert. Diese generierten Dateien werden nach der Testdurchführung des Consumers an den Provider weitergereicht. Zur Weiterreichung können verschiedene Methoden angewendet werden, die bereits in Abschnitt 3.6 beschrieben wurden. Pact bietet zusätzlich einen Pact-Broker an, der als zentraler Ablageort der Pact-Dateien fungiert [26]. Für die Beispielanwendung werden der einfachheitshalber die Pact-Dateien lediglich manuell kopiert und beim Catalog-Microservice eingefügt.

Der Provider hat nun die Aufgabe, die Consumer-Contracts auszuführen und zu verifizieren, dass der Provider die Verträge ordnungsgemäß einhält. Dazu wird im Catalog-Microservice die Testklasse aus Listing 4.12 angelegt. Wieder werden in den ersten drei Zeilen die Tests für Spring Boot vorbereitet. In Zeile vier wird die Testklasse für Pact als Provider markiert. Darunter befindet sich in Zeile fünf die Angabe des Ablageortes der Pact-Dateien. In Zeile neun wird ein Objekt für das `ProductRepository` vom Spring Framework injiziert. Dieses Objekt wird später für die Herstellung des jeweiligen Zustandes benötigt. Die Ausführung der Contract-Tests geschieht in der Methode `pactVerificationTestTemplate`. Diese Methode ist mit `TestTemplate` markiert, sodass für beliebig viele Contract-Tests eine jeweilige Testdurchführung stattfindet. Im Consumer-Contract des Order-Microservice wurde der Zustand `product exists` gefordert. Um die Herstellung dieses Zustandes kümmert sich die Methode `productExists` in Zeile 19 bis 25. Darin wird die Product-Id, welcher zuvor als Parameter vom Consumer übergeben wurde, verwendet, um dieses Produkt in der Datenbank zu erstellen. Weitere Methoden für die Herstellung eines Zustandes müssen entsprechend erstellt werden, wenn ein Consumer dies fordert.

Listing 4.12: Catalog-Provider führt Consumer-Contract-Tests aus

```
1  @SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
2  @ExtendWith(SpringExtension.class)
3  @ActiveProfiles("test")
4  @Provider("catalog")
5  @PactFolder("pacts")
6  class CatalogContractTest {
7
8    @Autowired
9    private ProductRepository productRepository;
10
11   @TestTemplate
12   @ExtendWith(PactVerificationSpringProvider.class)
13   void pactVerificationTestTemplate(
14       PactVerificationContext context) {
15     context.verifyInteraction();
16   }
17
```

```
18    @State("product exists")
19    void productExists(Map<String, String> parameters) {
20      UUID productId =
21          UUID.fromString(parameters.get("productId"));
22      Product product = new Product(productId, "ProductName",
23          "ProductDescription", 1.0);
24      productRepository.save(product);
25    }
26  }
```

Die gezeigten Contract-Tests kontrollieren die Kommunikation und die Interaktionen zwischen Microservices. Wenn ein Entwickler eine Änderung an einem Provider vornimmt, erhält dieser direkt Rückmeldung, ob die Änderung Einfluss auf andere Microservices hat. Wenn es zu inkompatiblen Änderungen kommt, können betroffenen Microservices über diese Änderungen informiert werden. Inhaltlich können Contract-Tests sowohl die Geschäftslogik als auch temporale Merkmale wie zum Beispiel die Überprüfung auf eine maximale Antwortzeit untersuchen. Oftmals werden in den Contract-Tests die gleichen Aspekte wie in den Komponententests abgedeckt. Dadurch werden diese Komponententests obsolet und sollten entfernt werden, um Duplikation der Tests zu vermeiden.

Contract-Tests finden für monolithische Anwendungen in der Regel keine Anwendung, da in einem Monolith inkompatible Änderungen direkt bei der Kompilierung auffallen. Contract-Tests sind gerade dann sehr nützlich, wenn verschiedene unabhängige Systeme miteinander interagieren, wie es bei einer Microservice-Architektur der Fall ist.

4.7 UI-Tests mit MockMvc

In UI-Tests werden Benutzerschnittstellen genauer unter die Lupe genommen. Bei der Beispielanwendung stellt lediglich das Microservice Webshop-Client eine Benutzerschnittstelle in Form einer Webseite bereit. Die Webseite des Webshops enthält mehrere Seiten. Dazu gehört unter anderem der Katalog, in welchem alle Produkte aufgelistet sind. In Listing 4.13 ist der dazugehörige Controller abgebildet, der bei einer Get-Anfrage auf den Pfad `/catalog` den Katalog zurückgibt. Die Controller-Klasse ist dafür verantwortlich, die Attribute für die Anzeige der Webseite zusammenzustellen und diese an Thymeleaf weiterzugeben. Thymeleaf ist eine serverseitige Template-Engine, um Vorlagen von Webseiten in Form von HTML-Dokumenten mit Daten zu befüllen und daraus die Webseite entsprechend für den Benutzer zu generieren [19]. In der Zeile 15 in Listing 4.13 wird dazu als Attribut die Produktliste übergeben, damit diese auf der Webseite angezeigt werden kann.

Listing 4.13: Die Klasse CatalogController stellt die Katalog-Seite bereit

```
1  @Controller
```

```
2  @RequestMapping("/catalog")
3  public class CatalogController {
4
5    private final CatalogService catalogService;
6
7    public CatalogController(CatalogService catalogService) {
8      this.catalogService = catalogService;
9    }
10
11   @GetMapping
12   public Mono<String> getCatalog(Model model) {
13     return catalogService.getCatalogProducts()
14         .doOnSuccess(products ->
15             model.addAttribute("catalogProducts", products))
16         .thenReturn("catalog")
17         .onErrorReturn("catalog-error");
18   }
19 }
```

Um den Controller zu testen, dass dieser die richtige Webseite von Thymeleaf rendern lässt und dabei die Attribute in korrekter Form übergibt, werden UI-Tests implementiert. Das Spring Boot Framework liefert passend hierzu einen `WebMvcTest`, womit der Controller isoliert getestet werden kann. In Listing 4.14 ist ein solcher Test aufgeführt. Dort wird in Zeile eins die Klasse als `WebMvcTest` markiert und dabei den zu testenden Controller angegeben. In den Zeilen vier und fünf wird ein Mock für die innere Abhängigkeit zum `CatalogService` erstellt. In den Zeilen sieben und acht wird ein `MockMvc` Objekt angelegt, welches durch das Spring Framework zugewiesen wird. Dieses Objekt wird dazu verwendet, um Anfragen direkt an den Controller zu stellen. Da der Controller auf ein reaktives Programmierparadigma mit WebFlux zurückgreift, wird ähnlich zu den in Abschnitt 4.5 gezeigten Komponententests ein `WebTestClient` verwendet. Da jedoch nur der Controller in Isolation getestet werden soll und nicht das gesamte Microservice, wird der `WebTestClient` mithilfe des Objektes von `MockMvc` in Zeile 15 erstellt. In der Testmethode in den Zeilen 18 bis 40 wird der Testfall geprüft, bei dem der Katalog mit einem Produkt angezeigt werden soll, das auf Lager ist. In den Zeilen 21 bis 26 wird deswegen ein Produkt erzeugt und dem Stub für die Abhängigkeit `CatalogService` mitgeteilt, dass das zuvor erzeugte Produkt in einer Liste zurückgegeben werden soll. Die Anfrage wird in den Zeilen 27 bis 29 auf den Controller ausgeführt. Anschließend wird die Antwort in den Zeilen 30 bis 32 ausgewertet. Die Auswertung erfolgt vorerst lediglich auf Protokollebene, da der Inhalt der Antwort eine HTML-Webseite darstellt und deswegen nicht vom `WebTestClient` ausgewertet werden kann. Um die Validierung der Webseite kümmert sich der `MockMvcWebTestClient`. Dazu wird in Zeile 34 das Ergebnis der Abfrage an diesen übergeben, um daraufhin die Inhalte zu prüfen. So wird in Zeile 35 getestet, ob die richtige Webseite geladen wurde und in den Zeilen 36 und 37 sichergestellt, dass die Produktliste in korrekter Form an die Webseite übergeben wurde.

Listing 4.14: UI-Test für den Katalog-Controller

```
@WebMvcTest(CatalogController.class)
class CatalogControllerTest {

  @MockBean
  private CatalogService catalogService;

  @Autowired
  private MockMvc mockMvc;

  private WebTestClient webTestClient;

  @BeforeEach
  void setUp() {
    webTestClient =
        MockMvcWebTestClient.bindTo(mockMvc).build();
  }

  @Test
  void getCatalog_ShowCatalog_IfProductInStock()
      throws Exception {
    Product product = new Product(UUID.randomUUID(),
        "ProductName", "ProductDescription", 1);
    CatalogProduct catalogProduct =
        new CatalogProduct(product, 1);
    given(catalogService.getCatalogProducts())
        .willReturn(Mono.just(List.of(catalogProduct)));
    EntityExchangeResult<byte[]> result = webTestClient.get()
        .uri("/catalog")
        .exchange()
        .expectStatus()
        .isOk()
        .expectBody()
        .returnResult();
    MockMvcWebTestClient.resultActionsFor(result)
        .andExpect(view().name("catalog"))
        .andExpect(model().attribute("catalogProducts",
            List.of(catalogProduct)));
  }
}
```

Neben dem Testen des Controllers mit einem `WebMvcTest` kann die Webseite auch näher mit anderen Tools wie zum Beispiel der Bibliothek `thymeleaf-testing` getestet werden, womit speziell der Renderprozess untersucht werden kann, dass die Daten auf der Webseite auch wie gewünscht dargestellt werden [27].

UI-Tests haben zum Ziel, die Benutzerschnittstellen zu testen. Dabei können UI-Tests

verschiedene funktionale Anforderungen, visuelle Gestaltung einer Benutzeroberfläche oder auch Benutzbarkeit der Benutzerschnittstelle untersuchen. Auch können temporale Aspekte geprüft werden und so zum Beispiel Laufzeittests für die Benutzerschnittstelle entwickelt werden.

Wenn UI-Tests ähnlich zu Unit-Tests in Isolation erstellt werden, dann bieten sie den Vorteil einer schnellen Ausführung und geben dem Entwickler frühzeitig Rückmeldung darüber, dass die Benutzerschnittstelle wie gewünscht funktioniert. UI-Tests können aber auch als End-To-End-Tests implementiert werden. Hierbei liefern die Tests eine hohe Zuversicht über den ordnungsgemäßen Ablauf des Gesamtsystems. Jedoch sind diese Art von Tests deutlich zeitaufwendiger in ihrer Testdurchführung.

Auch in einer monolithischen Anwendung können UI-Tests in gleicher Form entwickelt werden. Gerade UI-Tests, welche die Benutzerschnittstelle isoliert betrachten, werden genauso wie in einer Microservice-Architektur umgesetzt, indem abhängige Komponenten der Benutzerschnittstelle innerhalb des Monolithen durch Stubs ersetzt werden. Anders sieht es hingegen bei UI-Tests aus, die als End-To-End-Tests umgesetzt sind. Ein Monolith umfasst bereits das gesamte System, wodurch End-To-End-Tests direkt auf dem gestarteten Monolithen ausgeführt werden können. In einer Microservice-Architektur muss vor der Ausführung der UI-Test, die auf dem gesamten System ausgeführt werden, die gesamte Microservice-Landschaft gestartet und betrieben werden. Dieser Prozess ist deutlich komplexer und aufwendiger, wodurch sich auch die Testdurchführung verzögert.

4.8 End-To-End-Tests mit Selenium

End-To-End-Tests testen das Gesamtsystem, um damit die Funktionalität der gesamten Anwendung sicherzustellen. Dazu werden UI-Tests verwendet und das System als Black-Box betrachtet. Dementsprechend wird die in Abschnitt 4.1 vorgestellte Anwendung getestet. Um die End-To-End-Tests durchführen zu können, müssen zuerst alle Microservices sowie zusätzlich nötigen Dienste wie Datenbanken in einer separaten Testumgebung gestartet und betrieben werden. Die Testumgebung soll dabei der Produktivumgebung möglichst nahekommen. Um dies zu bewerkstelligen, ist es sinnvoll, die Infrastruktur reproduzierbar zu gestalten und zum Beispiel die Infrastruktur als Code in einem versionierten Repository abzulegen. Sobald alle Einheiten der Anwendung erfolgreich gestartet sind, können die End-To-End-Tests ausgeführt werden.

In Listing 4.15 ist ein solcher End-To-End-Test abgebildet. Sehr verbreitet ist Selenium zur Umsetzung von End-To-End-Tests, wobei beim abgebildeten Test der Selenium-Client für Java verwendet wird. Selenium unterstützt alle gängigen Webbrowser-Treiber wie Chrome, Firefox, Edge und so weiter. Im Beispiel wird in Listing 4.15 in der Methode `beforeAll` zunächst der Chrome-Treiber für alle Tests eingerichtet. In der zweiten Metho-

de `setUp` wird ein Objekt für den `ChromeDriver` jeweils für die einzelnen Tests erzeugt. Die Methode `tearDown` sorgt dafür, dass nach jedem einzelnen Test der Web-Treiber wieder ordnungsgemäß beendet wird, um Konflikten zwischen den Tests zu verhindern. In Zeile 21 geschieht schließlich die konkrete Testdurchführung.

Der Test orientiert sich an einer Customer Journey, in welcher ein Kunde einen Kaufprozess durchläuft. Der Kunde öffnet zunächst die Webseite des Webshops und schaut sich anschließend den Katalog an. Danach fügt der Kunde ein Produkt dem Warenkorb hinzu. Dann gibt er in der virtuellen Kasse die Kundeninformationen ein und schickt die Bestellung ab. Zum Schluss erwartet der Kunde eine Bestellbestätigung mit den Informationen der getätigten Bestellung. Diese Customer Journey wird in Listing 4.15 in der Methode `placeOrder` abgebildet. In Zeile 22 wird zuerst die Webseite des Webshops geöffnet. Kontrolliert wird das Öffnen der Webseite in den Zeilen 23 und 24, wobei der Titel der Webseite auf die Korrektheit geprüft wird. Zusätzlich wird in den Zeilen 25 und 26 der Link zum Katalog aus der Navigationsbar erfasst und in Zeile 27 geprüft, ob dieses Element existiert. Anschließend wird in Zeile 28 dieser Link angeklickt. Das Anklicken übernimmt Selenium automatisiert. Da der Link auf eine weitere Seite leitet und diese Weiterleitung etwas Zeit benötigt, bis die Seite geladen hat, wird in Zeile 30 ein `WebDriverWait` mit einem Time-out von 10 Sekunden angelegt. In Zeile 32 wird dieser `WebDriverWait` dann aufgerufen, um beim erfolgreichen Laden der Webseite ein Button-Element zu ermitteln, der ein Produkt zum Warenkorb hinzufügt. Der Text des Button-Elementes wird danach geprüft und anschließend die Methode `submit` aufgerufen, um einen Klick auf das Schaltflächenelement auszulösen. Selbes passiert in den Zeilen 39 bis 41 mit dem Checkout-Button. Die Kundeninformationen werden mithilfe von Selenium mit Testdaten gefüllt, indem die Methode `sendKeys` aufgerufen wird. Nach der Eingabe der Kundeninformationen wird die Bestellung abgeschickt und in der Bestellinformation nochmals die Daten auf ihre Richtigkeit geprüft. Der Übersichtlichkeit wurde der End-To-End-Test in Listing 4.15 vereinfacht. Üblicherweise werden weitere Behauptungen ergänzt, um den Inhalt und das Verhalten der Webseite genauer zu testen.

Listing 4.15: End-To-End-Test des Webshops mit Selenium, um eine Bestellung aufzugeben

```
1  class WebshopClientEnd2EndTest {
2
3    private WebDriver webDriver;
4
5    @BeforeAll
6    static void beforeAll() {
7      WebDriverManager.chromedriver().setup();
8    }
9
10   @BeforeEach
11   void setUp() {
12     webDriver = new ChromeDriver();
13   }
14
```

```
15    @AfterEach
16    void tearDown() {
17      webDriver.quit();
18    }
19
20    @Test
21    void placeOrder() {
22      webDriver.get("http://localhost:8080");
23      assertThat(webDriver.getTitle())
24          .isEqualTo("Webshop");
25      WebElement catalogLink = webDriver.findElement(
26          By.linkText("Catalog"));
27      assertThat(catalogLink.isDisplayed()).isTrue();
28      catalogLink.click();
29
30      WebDriverWait wait = new WebDriverWait(webDriver, 10);
31
32      WebElement addToCartButton = wait.until(
33          ExpectedConditions.presenceOfElementLocated(
34              By.cssSelector("tr:nth-child(2) button")));
35      assertThat(addToCartButton.getText())
36          .isEqualTo("Add to cart");
37      addToCartButton.submit();
38
39      WebElement checkoutButton =
40          webDriver.findElement(By.cssSelector("button"));
41      checkoutButton.submit();
42
43      WebElement firstNameInput = webDriver.findElement(
44          By.id("firstName"));
45      firstNameInput.sendKeys("SeleniumCustomerFirstName");
46      WebElement lastNameInput = webDriver.findElement(
47          By.id("lastName"));
48      lastNameInput.sendKeys("SeleniumCustomerLastName");
49      WebElement addressInput = webDriver.findElement(
50          By.id("address"));
51      addressInput.sendKeys("SeleniumCustomerAddress");
52
53      WebElement orderButton = webDriver.findElement(
54          By.cssSelector("button"));
55      orderButton.submit();
56
57      WebElement firstNameLabel = webDriver.findElement(
58          By.id("firstName"));
59      assertThat(firstNameLabel.getText())
60          .isEqualTo("SeleniumCustomerFirstName");
61      WebElement lastNameLabel = webDriver.findElement(
62          By.id("lastName"));
63      assertThat(lastNameLabel.getText())
```

```
64          .isEqualTo("SeleniumCustomerLastName");
65      WebElement addressLabel = webDriver.findElement(
66          By.id("address"));
67      assertThat(addressLabel.getText())
68          .isEqualTo("SeleniumCustomerAddress");
69    }
70  }
```

End-To-End-Tests geben eine hohe Zuversicht darüber, dass die Endanwendung problemlos funktioniert. So fallen Fehler direkt auf, die dazu führen würden, dass wichtige Teile des Systems nicht richtig arbeiten. Dadurch werden unter anderem auch Fehler in der Kommunikation zwischen Microservices bemerkt, die das erfolgreiche Durchlaufen eines Customer Journeys behindern. End-To-End-Tests sind in der Testdurchführung relativ langsam, da gerade bei einer Microservice-Architektur das Gesamtsystem sehr komplex ausfallen kann. Die Gefahr besteht hierbei, dass ein Entwickler erst sehr spät die Rückmeldung der Testergebnisse erhält und sich so der Entwicklungsprozess verzögert. Zudem sind die Tests sehr aufwendig in der Wartung, da Änderungen in vielen verschiedenen Microservices früher oder später in fehlschlagenden End-To-End-Tests resultieren und dann die Tests ebenso angepasst werden müssen. Aufgrund der Nachteile von End-To-End-Tests empfiehlt es sich, nur wichtige Customer Journeys abzubilden, um den Aufwand der End-To-End-Tests zu reduzieren, aber dennoch eine hohe Zuversicht über die Funktionsweise des Gesamtsystems zu erhalten.

In Monolithen können End-To-End-Tests ebenso eingesetzt werden. Ein Monolith profitiert davon, dass dieser bereits das Gesamtsystem bildet und alleinstehend im Vergleich zu einer Microservice-Landschaft schneller startet. Durch die zügigere Bereitstellung des Gesamtsystems eines Monolithen dauert die Testdurchführung der End-To-End-Tests bei Monolithen nicht so lange.

5 Fazit

5.1 Zusammenfassung

Das Ziel dieser Bachelorarbeit war es, automatisierte Teststrategien zur Verbesserung der Servicequalität einer Microservice-Architektur aufzudecken. Dazu wurden verschiedene Teststrategien für Microservices herausgearbeitet und die Implementierung in einer Beispielanwendung vorgestellt.

Zuerst wurde die Wichtigkeit der Automatisierung der Tests unterstrichen. Automatisierte Tests sind im Vergleich zu mühsamen manuellen Tests deutlich schneller und weniger fehleranfällig in der Testdurchführung. Umgesetzt werden automatisierte Tests mithilfe einer Continuous-Integration Pipeline, die nach jeder Quellcodeänderung automatisiert zunächst den Quellcode kompiliert und anschließend die Tests ausführt. Durch die Automatisierung erhält der Entwickler frühzeitig Rückmeldung über die Testergebnisse.

Als Nächstes wurde die Testpyramide betrachtet, die vor allem in monolithischen Anwendungen eingesetzt wird. Die Testpyramide teilt Softwaretests in Prüfebenen ein. Auf der untersten Ebene befinden sich die Unit-Tests, die eine atomare Programmeinheit testen. Auf der zweiten Ebene sitzen die Service-Tests, die das Zusammenspiel der Komponenten prüfen, und auf der obersten Ebene befinden sich die UI-Tests, die das Gesamtsystem testen. Die Pyramidenform spiegelt die Testverteilung wider, sodass viele Unit-Tests, einige Service-Tests und wenige UI-Tests entwickelt werden sollen. Häufig wird auf die Testpyramide aufgebaut und unter anderem die Service-Tests in Komponententests, Integrationstests und API-Tests weiter unterteilt. Auch wird häufig die Testpyramide mit manuellen Tests erweitert, die bei Bedarf ausgeführt werden.

Daraufhin wurden die Unit-Tests untersucht. Unit-Tests prüfen atomare Bausteine des Programms. Ein solcher Baustein kann eine einzelne Funktion, aber auch eine gesamte Klasse oder mehrere Klassen, sein. Unit-Tests werden in der Regel in derselben Programmiersprache wie das getestete Programm entwickelt. Konstruiert werden Unit-Tests als White-Box-Tests oder als Black-Box-Tests. Bei White-Box-Tests wird die innere Struktur des Programms analysiert und ein Kontrollflussgraph oder ein Datenflussgraph abgeleitet, womit Testfälle in Unit-Tests implementiert werden. Bei Black-Box-Tests werden die Anforderungen an das Programm hergenommen, um daraus Testfälle zu erzeugen. Unit-Tests lassen sich in Sociable Unit-Tests und Solitary Unit-Tests einteilen. Sociable

Unit-Tests kontrollieren die Geschäftslogik, Solitary Unit-Tests fokussieren sich auf die Interaktionen mit Abhängigkeiten. In Unit-Tests werden Abhängigkeiten mit Stubs oder Mocks simuliert, um äußere Einflüsse zu vermeiden.

Danach wurden Integrationstests behandelt. Integrationstests beschäftigen sich mit dem Zusammenspiel von externen Abhängigkeiten. Dabei lassen sich die Tests in Persistent-Integrationstests und Gateway-Integrationstests aufteilen. Persistent-Integrationstests kümmern sich um die erfolgreiche Anbindung des Programms mit einem Datenspeicher. Gateway-Integrationstests kontrollieren die Integration von externen Diensten und überprüfen unter anderem das Netzwerkprotokoll auf ihre Richtigkeit und beobachten das Verhalten des Programms bei einem Fehler oder Ausfall der externen Abhängigkeit. Um Integrationstests durchzuführen, wird das Microservice sowie die äußeren Abhängigkeiten lokal gestartet. Die Abhängigkeiten werden hierbei mit Test-Double ersetzt.

Anschließend wurden die Komponententests inspiziert. Komponententests beziehen das gesamte Microservice mit ein und prüft das Microservice auf funktionale und temporale Kriterien. Externe Abhängigkeiten werden bei Komponententests mit Test-Double ersetzt, um das Microservice isoliert zu testen. Komponententests lassen sich weiter in In-Process und Out-Of-Process Komponententests einteilen. Bei In-Process Komponententests werden Test-Double direkt im Arbeitsspeicher geladen. Hierfür muss das Programm einen Test-Modus unterstützen, um die Test-Double einsetzten zu können. Bei Out-Of-Process Komponententests wird das Microservice unverändert getestet und die Abhängigkeiten von außen bereitgestellt.

Nachfolgend wurden Contract-Tests vorgestellt. Contract-Tests eigenen sich, um die Kompatibilität zwischen Microservices und deren Schnittstellen sicherzustellen. Hierzu wird der Nutzer eines Microservices als Consumer und das Microservice, welche die API bereitstellt, als Provider bezeichnet. Der Consumer definiert in einem Consumer-Contract, wie er den Provider einsetzt und erstellt dazu passende Consumer-Contract-Tests. Diese Consumer-Contract-Tests werden mit Consumer-Contract-Tests von anderen Consumern als Consumer-driven Contract-Tests zusammengefasst und dem Provider übergeben. Der Provider führt schließlich die Contract-Tests aus und verifiziert die Korrektheit seiner Schnittstelle. Bei einer Änderung, die zu einem fehlschlagenden Contract-Test führt, ist direkt bekannt, bei welchem Consumer die Änderung Probleme verursachen wird.

Daraufhin wurden UI-Tests betrachtet. UI-Tests untersuchen die Benutzerschnittstellen des Programms. Zu den Benutzerschnittstellen gehören nicht nur die grafische Benutzeroberflächen, sondern auch REST-Schnittstellen, die für den Benutzer zugänglich sind. UI-Tests können die Benutzerschnittstelle sowohl in Isolation testen, aber auch auf das Gesamtsystem angewendet werden. Wenn die UI-Tests in Isolation testen, werden sie häufig als Unit-Tests implementiert, wobei Abhängigkeiten durch Stubs ersetzt werden. Beim Testen der Benutzerschnittstelle am Gesamtsystem handelt es sich um End-To-End-Tests.

Folglich wurden End-To-End-Tests thematisiert. End-To-End-Tests testen das Gesamtsystem auf ihre korrekte Funktionsweise. Dazu werden alle Microservices sowie alle abhängigen Dienste in einer Testumgebung gestartet. Für die Konstruktion der End-To-End-Tests werden meistens Customer Journeys in Tests abgebildet, wobei das System als Black-Box betrachtet wird. Dadurch gewähren End-To-End-Tests eine hohe Sicherheit darüber, dass das Gesamtsystem einwandfrei arbeitet.

Die vorgestellten Teststrategien für eine Microservice-Architektur wurden anschließend in einer Beispielanwendung implementiert. Bei der Beispielanwendung handelt es ich um einen einfachen Webshop, bei dem Produkte bestellt werden können. Die Beispielanwendung besteht aus fünf Microservices, die in Spring Boot programmiert wurden. Die Unit-Tests wurden mit JUnit umgesetzt, wobei Stubs mithilfe von Mockito erzeugt wurden und AssertJ zum Aufstellen von Behauptungen verwendet wurde. Die Persistent-Integrationstests wurden einmal mithilfe der Testerweiterung von Spring Boot für die Java Persistence API erstellt, wobei die Datenbank mit einer In-Memory-Datenbank ausgetauscht wurde, und einmal mit Testcontainers umgesetzt, welche die Datenbank lokal zum Testen in Docker startet. Um bei den Gateway-Integrationstests die externen Abhängigkeiten durch Test-Double zu ersetzen, wurde WireMock verwendet. Bei den Komponententests kam WebFlux zum Einsatz, um REST-Anfragen an das Microservice zu stellen. Bei den Komponententests wurden der Datenspeicher wieder durch eine In-Memory-Datenbank und externe Dienste mit WireMock ersetzt. Contract-Tests wurden mit Pact erstellt. Dazu wurden zuerst Consumer-Contract mit Pact definiert, die eine Pact-Datei generieren. Diese Datei wurden anschließend an den Provider weitergegeben, der die Contract-Tests ausführt und so die API validiert. UI-Tests, welche die Benutzeroberfläche isoliert testen, wurden mit MockMvc implementiert. End-To-End-Tests haben schlussendlich die gesamte Anwendung getestet, wobei Selenium zum Einsatz kam. Selenium führt automatisiert Aktionen auf einer Webseite aus und validiert dabei die Weboberfläche.

5.2 Ausblick

Die vorgestellten Teststrategien für Microservice-Architekturen lassen sich auch auf andere Architekturen übertragen. Gerade Unit-Tests können für jeden beliebigen Quelltext entwickelt werden und sind unabhängig von der umfassenden Architektur. Auch Persistent-Integrationstests können überall dann eingesetzt werden, sobald eine Datenspeicherung erfolgt. Sollte eine Anwendung eine externe Abhängigkeit besitzen, können hierfür zudem Gateway-Integrationstests programmiert werden. Contract-Tests können nicht nur in Microservice-Architekturen verwendet werden, sondern auch für andere serviceorientierte Architekturen zum Einsatz kommen, um damit die Integration zwischen verschiedenen Diensten zu verbessern und Fehler in der Kommunikation im Vorfeld zu entdecken.

In der Arbeit wurde der Fokus auf Teststrategien für Microservice-Architekturen gelegt. Es bleibt ferner offen, wie diese Teststrategien genau in einer Continuous-Integration / Continuous-Delivery Pipeline automatisiert werden können und wie die Implementierung dieser Automatisierung aussieht. Dabei wäre auch interessant, wie vor allem die End-To-End-Tests automatisiert werden, bei denen die Infrastruktur als Code vorliegt, da sich die Bereitstellung der gesamten Anwendung bei einer Microservice-Landschaft als komplex herausstellt. Eine weitere Möglichkeit zum Testen von Software bieten künstliche Intelligenzen, die unter anderem Testfälle generieren können. Hierzu wäre wissenswert, inwieweit künstliche Intelligenzen die in dieser Arbeit vorgestellten Teststrategien ergänzen und erweitern können. Des Weiteren könnte untersucht werden, welche Teststrategien in der Produktionsumgebung angewendet werden können, um beispielsweise verschiedene Versionen eines Microservices zu testen und dabei das Benutzerverhalten zu analysieren. Eine weitere Methode, um die Servicequalität zu verbessern, ist das Monitoring einer Anwendung. Hierbei könnte genauer untersucht werden, wie sich eine Microservice-Architektur überwachen lässt, um Fehler in der Produktionsumgebung möglichst schnell ausfindig zu machen und zu beheben.

Literaturverzeichnis

[1] P. B. Kruchten, "The 4+1 View Model of architecture," *IEEE Software*, vol. 12, no. 6, S. 42–50, 1995.

[2] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*. Pearson Education, 2010.

[3] D. Kolesnikov, "Using Microservices to Power Fashion Search and Discovery," *Zalando Engineering Blog*. Februar 2017. [Online]. Available: https://engineering.zalando.com/posts/2017/02/using-microservices-to-power-fashion-search-and-discovery.html (Abrufdatum: 01.09.2021).

[4] F. S. Minguel, "The Netflix Cosmos Platform," *Netflix TechBlog*. März 2021. [Online]. Available: https://netflixtechblog.com/the-netflix-cosmos-platform-35c14d9351ad (Abrufdatum: 01.09.2021).

[5] J. Lewis und M. Fowler, "Microservices Guide," 2014. [Online]. Available: https://martinfowler.com/microservices/ (Abrufdatum: 01.09.2021).

[6] G. D. Everett, "The Value of Software Testing to Business: The Dead Moose on the Table," *testing experience*, Juni 2008.

[7] "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, S. 1–84, 1990.

[8] F. Witte, *Testmanagement und Softwaretest: Theoretische Grundlagen und praktische Umsetzung*. Springer, 2019.

[9] D. W. Hoffmann, *Software-Qualität*. Springer-Verlag, 2013.

[10] "IEEE Recommended Practice for Architectural Description for Software-Intensive Systems," *IEEE Std 1471-2000*, S. 1–30, 2000.

[11] J. Goll, *Architektur- und Entwurfsmuster der Softwaretechnik*, 2nd ed. Springer Vieweg, 2014.

[12] S. Newman, *Building Microservices*, 2nd ed. O'Reilly Media, Inc., 2021.

[13] V. Vernon, *Domain-Driven Design Distilled*. Addison-Wesley Professional, 2016.

[14] J. Willett, "The Evolution of the Testzing Pyramid," September 2016. [Online]. Available: https://www.james-willett.com/the-evolution-of-the-testing-pyramid/ (Abrufdatum: 14.12.2021).

[15] E. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*. dpunkt. verlag, 2018.

[16] T. Clemson, "Testing Strategies in Microservice Architecture," November 2014. [Online]. Available: https://martinfowler.com/articles/microservice-testing/ (Abrufdatum: 04.02.2022).

[17] H. Vocke, "The Practical Test Pyramid," Februar 2018. [Online]. Available: https://martinfowler.com/articles/practical-test-pyramid.html (Abrufdatum: 05.02.2022).

[18] "Spring Boot," *Spring*. [Online]. Available: https://spring.io/projects/spring-boot (Abrufdatum: 27.01.2022).

[19] "Thymeleaf," *Thymeleaf*. [Online]. Available: https://www.thymeleaf.org/ (Abrufdatum: 27.01.2022).

[20] "Spring Data JPA," *Spring*. [Online]. Available: https://docs.spring.io/spring-data/data-jpa/docs/2.6.3/reference/html/ (Abrufdatum: 27.01.2022).

[21] M. Keith, M. Schincariol, und M. Nardone, *Pro JPA 2 in Java EE 8: An In-Depth Guide to Java Persistence APIs*. Apress, 2018.

[22] "Spring Cloud netflix," *Spring*. [Online]. Available: https://spring.io/projects/spring-cloud-netflix (Abrufdatum: 31.01.2022).

[23] C. Tudose, *JUnit in Action*, third edition ed. Manning Publications, 2020.

[24] "Web on Reactive Stack," *Spring*. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html (Abrufdatum: 29.01.2022).

[25] "pact-jvm-consumer-junit5," *Pact Docs*. [Online]. Available: https://docs.pact.io/ (Abrufdatum: 14.02.2022).

[26] "Sharing Pacts with the Pact Broker," *Pact Docs*. [Online]. Available: https://docs.pact.io/getting_started/sharing_pacts (Abrufdatum: 14.02.2022).

[27] “Thymeleaf Testing Library,” *GitHub*. [Online]. Available: https://github.com/thymeleaf/thymeleaf-testing (Abrufdatum: 16.02.2022).

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Bachelorarbeit mit dem Thema „Automatisierte Teststrategien für Microservice-Architekturen“ selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet habe.

Ort, Datum Unterschrift