HAW HAMBURG

BACHELOR THESIS
Arseny Yaremenko

Realisierung einer Microservice-Architektur am Beispiel einer E-Commerce-Anwendung für Musikproduktionstemplates

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Arseny Yaremenko

Realisierung einer Microservice-Architektur am Beispiel einer E-Commerce-Anwendung für Musikproduktionstemplates

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 19. August 2024

Arseny Yaremenko

Thema der Arbeit

Realisierung einer Microservice-Architektur am Beispiel einer E-Commerce-Anwendung für Musikproduktionstemplates

Stichworte

Microservices, E-Commerce, Software Engineering, Containerization

Kurzzusammenfassung

Die vorliegende Arbeit stellt eine Microservice-Architektur für eine E-Commerce-Anwendung vor, die speziell für Musikproduktionstemplates entwickelt wurde. Sie durchläuft dabei sämtliche Schritte des Software-Engineering-Prozesses: von der Anforderungsermittlung über die Konzeption der Anwendung bis hin zur Realisierung mit dem Ziel eines funktionsfähigen Prototyps. Im Rahmen einer Evaluierung werden, neben der Erfüllung der Anforderungen, die Vor- und Nachteile dieser Architektur für das gegebene Problem untersucht und alternative Ansätze besprochen. Zum Abschluss werden mögliche Verbesserungen des Prototyps vorgeschlagen.

Arseny Yaremenko

Title of Thesis

Implementation of a microservice architecture using the example of an e-commerce application for music production templates

Keywords

Microservices, E-Commerce, Software Engineering, Containerization

Abstract

This work introduces a microservice architecture for an e-commerce application specifically designed for music production templates. It follows all steps of the software

engineering process: from gathering requirements to designing and implementing the application, with the goal of creating a functional prototype. In addition to evaluating how well the requirements are met, this work examines the advantages and disadvantages of this architecture for the given problem and discusses alternative approaches. Finally, potential improvements to the prototype are suggested.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

DSGVO Datenschutz-Grundverordnung.

JPA Java Persistence API.

JVM Java Virtual Machine.

MVC Model-View-Controller.

MVVM Model-View-ViewModel.

VM Virtuelle Maschine.

VPC Virtual Private Cloud.

1 Einleitung

1.1 Problemstellung

SaaS-Onlineshops wie Shopify bieten wenig Möglichkeit der Konfigurierung, wenn es um den Verkauf von Templates für Musikproduktionen geht. Templates beschreiben Projekt-Dateien von Musikproduktionen, welche Kunden die Möglichkeit bietet, in die Spuren und Effekte einer Musikproduktion zu blicken. Will man sich von der Konkurrenz abheben, benötigt man hier eine maßgefertigte Lösung mit speziellen Features, wie z. B. einer Kommentarfunktion auf allen Einzelspuren der Templates. Somit können Kunden sich schnell vom Produkt überzeugen, da sie anhand aller Einzelspuren genau sehen können, was sie bekommen. Dennoch muss die Anwendung selbst hochverfügbar und schnell anpassungsfähig sein, um gegen spezialisierte Anbieter anzukommen.

1.2 Motivation

Da ich mich hobbymäßig selbst mit Musikproduktion beschäftige und solch einen Onlineshop erstellen möchte, habe ich mir schon einige Konkurrenten angeschaut sowie zahlreiche Templates zu Lernzwecken erworben.

Oft ist ein Problem, dass der Inhalt solcher Templates nicht transparent dargestellt wird. Dabei ist schon mal vorgekommen, dass der Syntheseweg bestimmter wichtiger Sounds gar nicht gezeigt wird, sondern nur als Audiospur vorliegt. Würde man sich als Kunde vorher alle Spuren anhören und kommentieren können, würde es die Kaufentscheidung des Kunden und das Produkt selbst positiv beeinflussen. Viele negative Kommentare einer Spur könnten beispielsweise den Verkäufer dazu bewegen, Anpassungen vorzunehmen oder sich mit den Kunden bezüglich seiner Mixing-Entscheidungen auszutauschen.

1.3 Zielsetzung

Ziel ist die Realisierung eines lauffähigen Online-Shop-Prototyps für Musikproduktionstemplates auf Grundlage einer Microservice-Architektur in der Cloud. Dabei werden sämtliche Techniken sowie Vor- und Nachteile, die mit einer Microservice-Architektur einhergehen, evaluiert und der Weg zur Umsetzung des Prototyps erläutert.

1.4 Struktur der Arbeit

Die Arbeit ist in sechs Kapiteln gegliedert. Zunächst behandelt Kapitel 1 die Problemstellung sowie die Zielsetzung der Arbeit. Darauf aufbauend folgt in Kapitel 2 eine umfassende Anforderungsanalyse, in der die grundlegenden Anforderungen und Randbedingungen des Systems untersucht werden. Hierbei werden auch erste Diagramme wie das Systemkontext- und das Domänenklassendiagramm erstellt.

Kapitel 3 widmet sich dem Systemdesign und beleuchtet die Architektur sowie die technischen Details des Systems. Anhand des arc42-Templates werden hier verschiedene Sichtweisen des Systems betrachtet und alle Entscheidungen vom Architekturstil bis zur Bereitstellung spezifiziert und begründet.

In Kapitel 4 wird die konkrete Umsetzung des Systems vorgestellt. Dies umfasst die Auswahl der Backend- und Frontend-Technologien sowie die Implementierung der Microservices. Die Herausforderungen und Lösungsansätze, die während der Entwicklung auftraten, werden ebenfalls erläutert.

Das fünfte Kapitel konzentriert sich auf die Evaluation des Systems. Hier wird der Prototyp hinsichtlich der Erfüllung der definierten Anforderungen geprüft und die umgesetzten Lösungen kritisch bewertet.

Abschließend gibt das letzte Kapitel einen Ausblick auf zukünftige Entwicklungen und mögliche Verbesserungen, die für den Produktionsbetrieb relevant sein könnten.

2 Anforderungsanalyse

Die erste Phase des Softwareentwicklungsprozesses ist die Anforderungsanalyse. In Zusammenarbeit mit den Stakeholdern werden Anforderungen an das System gesammelt und eine Systemspezifikation ausgearbeitet. [Som18, S.5] Diese Spezifikation, oder auch Pflichtenheft genannt, definiert alle funktionalen und nicht-funktionalen Anforderungen, welche das System leisten soll, sowie die Randbedingungen, unter denen das System operiert. Damit wird eine klare und rechtlich wirksame Vereinbarung zwischen Entwicklern und Kunden getroffen, die am Ende durch einen Validierungsprozess auch überprüfbar sein soll. Im folgenden Kapitel wird genau solch eine Spezifikation ausgearbeitet, um die Grundlage für den weiteren Entwicklungsprozess zu schaffen.

2.1 Funktionsweise

Die Webanwendung ermöglicht es dem Kunden, seine Musikproduktionstemplates hochzuladen und Käufern online zur Verfügung zu stellen. Die jeweilige Produktseite eines Templates enthält Metadaten zum Template, ein Bild, eine Gesamtspur der Musikproduktion sowie alle Einzelspuren. Das heißt, dass all diese Informationen vorher in einem Uploadformular angegeben und die jeweiligen Spuren hochgeladen werden müssen. Auf der Homepage werden dann alle Templates als Bilder angezeigt, welche durch einen Klick auf die jeweilige Verkaufsseite des Templates führen.

Jede Spur der Produktseite kann vom Nutzer oder Kunden kommentiert werden. Die Darstellung der Kommentare erfolgt ähnlich wie bei SoundCloud: Jeder Kommentar eines Nutzers wird zuerst durch ein Nutzerbild auf der Audiospur angezeigt, wobei die Position des Bildes durch die vergangene Zeit der Audiospur bestimmt wird. Fährt man mit der Maus über das Bild, öffnet sich ein Fenster, welches den Kommentar anzeigt. Die Bilder selbst werden ähnlich wie bei MS-Teams als Initialen der Nutzer wiedergegeben.

Jede Produktseite enthält einen *Add to Cart-Button*, welcher das Produkt in den virtuellen Warenkorb des Nutzers hinzufügt. Auf der Seite des Warenkorbs kann der Nutzer alle seine Produkte bearbeiten und auf die Checkout-Seite gelangen, wo ein Kauf mit den dafür notwendigen Kundeninformationen durchgeführt werden kann. Zusammengefasst wird eine klassische E-Commerce-Anwendung entwickelt, welche aber hinsichtlich der Produktseite den Nutzern und Kunden zusätzliche Features bereitstellt.

2.2 Stakeholder

Stakeholder sind die Personen oder Organisationen, die direkten oder indirekten Einfluss auf die Anforderungen des Systems haben. [PR15, S.4] Um also möglichst alle notwendigen Anforderungen abzudecken, müssen zuerst die wichtigsten Stakeholder ermittelt werden:

Die beiden wichtigsten Stakeholder sind der Kunde der Anwendung und die Käufer der Templates. Beide Rollen sind Musikproduzenten und werden unter anderem maßgeblichen Einfluss auf zukünftige Versionen der Anwendung haben. Da der Prototyp erst mal nur für einen Kunden und nicht als Massenprodukt für viele Kunden gedacht ist, wird hier ein Produktmanager nicht berücksichtigt. Ein weiterer wichtiger Stakeholder sind die Entwickler. Diese setzen die funktionalen und nicht-funktionalen Anforderungen um und beeinflussen die Gesamtqualität des Produkts. Der Begriff des Entwicklers ist weit gefasst, weshalb man diesen Stakeholder je nach Aufgabentyp wie beispielweise Tester und Operator in Unterrollen aufteilen könnte. Da dieses Projekt aber erst mal nur von einer Person entwickelt wird, werden auf Untergruppen verzichtet. Die letzten nennenswerten Stakeholder sind die IT-Recht-Spezialisten. Sie stellen als Berater sicher, dass die Anwendung rechtskonform in Betrieb genommen wird. Dies ist nötig, da der Rechtekatalog für eine E-Commerce-Anwendung komplex ist und sich stets wandelt. Außerdem könnten die hohen Bußen bei Verstößen für kleinere Unternehmen existenzbedrohend sein.

Zusammengefasst erhalten wir folgende Stakeholder, welche die Anforderungen mitbestimmen:

- Kunde der Anwendung
- Nutzer der Anwendung bzw. Käufer der Produkte des Kunden

- Entwickler
- IT-Recht-Spezialisten

2.3 Systemkontext

Im Rahmen des Requirements Engineering wird der Systemkontext bestimmt. Der Systemkontext zeigt, welche Aspekte mit dem eigenen System interagieren und legt somit eine Kontextgrenze fest, die für die Definition der Anforderungen relevant ist. Aspekte wären beispielsweise Stakeholder oder Fremdsysteme, mit denen das eigene System agiert. [PR15, S.13-16]

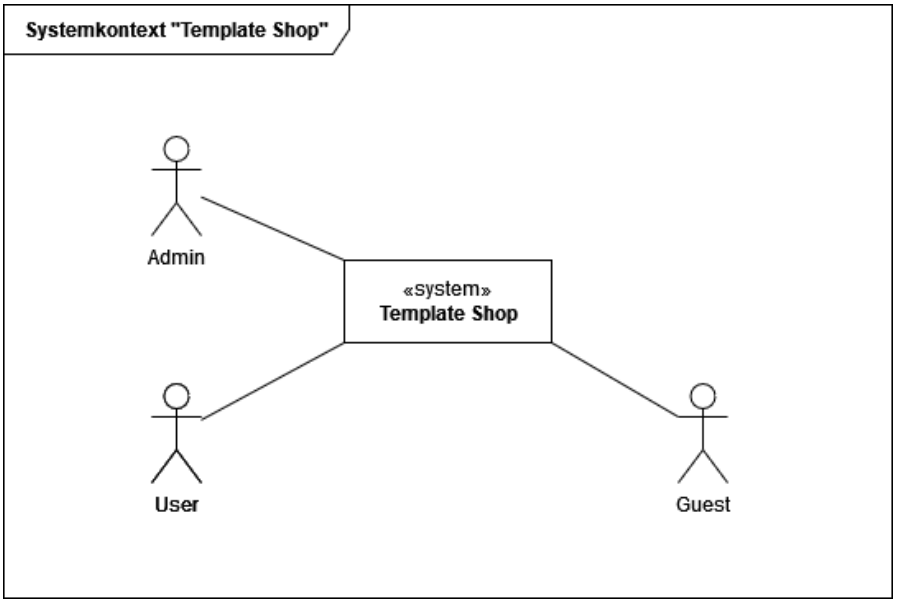

Abbildung 2.1: UML Systemkontextdiagramm des Template Shops

Abbildung 2.1 zeigt das Systemkontextdiagramm unseres Systems, welches wir als Template Shop bezeichnen. Dieses System interagiert mit drei Akteuren: dem *Guest*, dem *User* und dem *Admin*. Ein *Guest* ist ein Nutzer, der die Anwendung aufruft, aber noch kein eigenes Konto besitzt und ohne Kundenkonto Käufe tätigen kann. Der *User* ist ein registrierter *Guest*, der zusätzlich die Audiospuren der Templates kommentieren kann. Der *Admin* ist der Kunde des Systems, der seine Produkte für den Template Shop anlegt und den *Guests* sowie *Usern* zum Verkauf anbietet. Fremdsysteme wie ein Payment- oder

E-Mail-Service-Provider werden nicht dargestellt, da sie in der aktuellen Version dieses Prototyps nicht verwendet werden.

2.4 Anforderungen

In Zusammenarbeit mit den Stakeholdern wurden die nötigen Anforderungen an das System ausgearbeitet. Dabei wurden funktionale Anforderungen und nicht-funktionale Anforderungen definiert: Während funktionale Anforderungen eine dem Benutzer zur Verfügung gestellte Funktionalität des Systems beschreiben, beziehen sich nicht-funktionale Anforderungen auf Eigenschaften des Systems, die nicht von funktionalen Anforderungen abgedeckt werden. [PR15, S.8-9]

2.4.1 Funktionale Anforderungen

Um die funktionalen Anforderungen übersichtlich und verständlich darzustellen, wurde ein Use-Case-Diagramm erstellt. Das Use-Case-Diagramm baut auf dem Systemkontext-Diagramm auf und stellt alle Funktionalitäten des Systems mit den Interaktionspartnern bzw. Aspekten in Beziehung. [PR15, S.38]

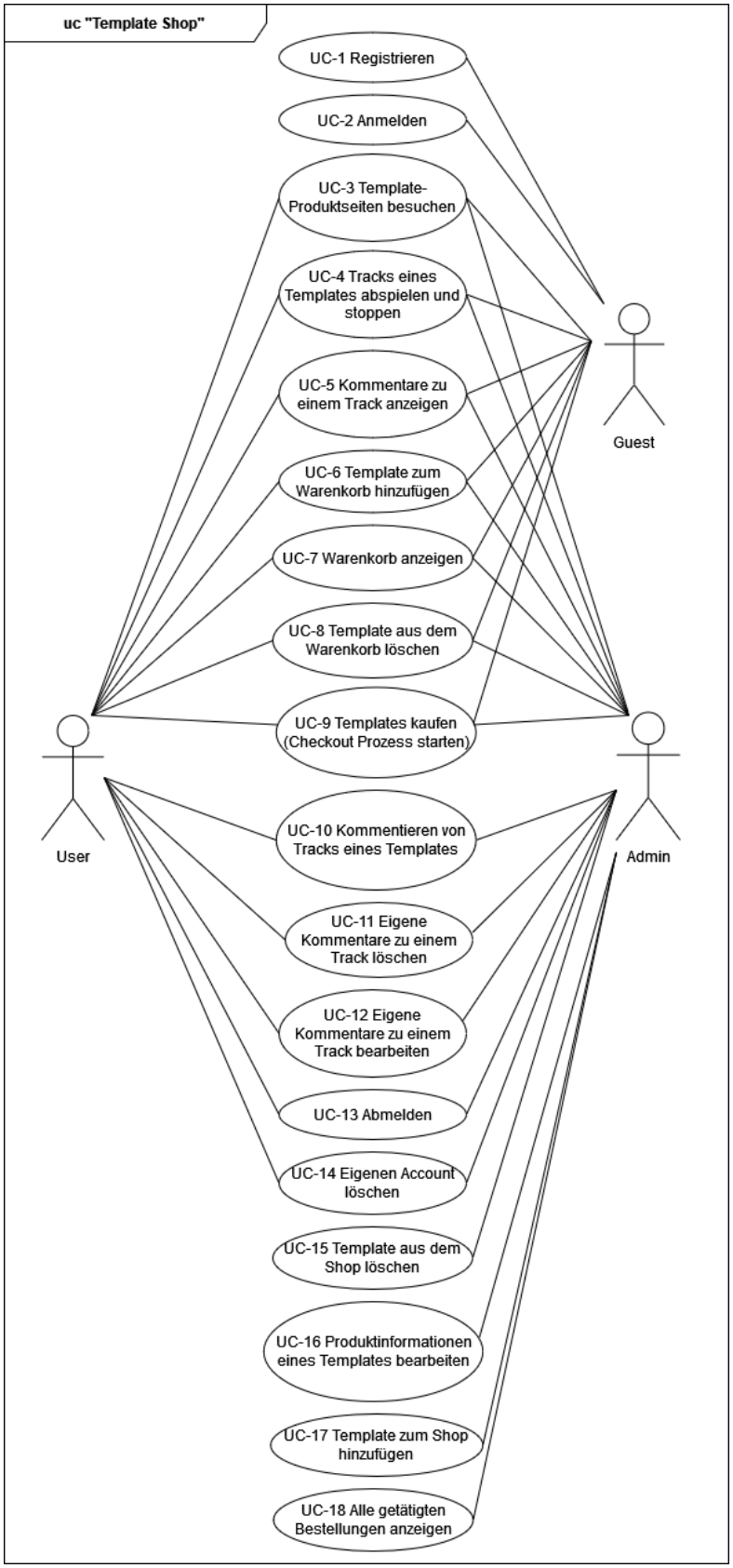

Abbildung 2.2: UML Use-Case Diagramm für den Template Shop

Abbildung 2.2 stellt das Use-Case-Diagramm für den Template Shop dar. Daraus werden die einzelnen Use Cases und deren Beziehung zu den Akteuren im System ersichtlich. Da ein Use Case selbst jedoch wenig Details zu einer Funktionalität bietet, ist es nötig, diese zu spezifizieren. Beispielsweise wäre es sinnvoll, Haupt- und Alternativabläufe sowie Vor- und Nachbedingung der Funktionalität detailliert zu beschreiben, um Klarheit im Entwicklungsprozess zu schaffen. [PR15, S.72-73] Die Spezifikation der 18 Use Cases findet sich aufgrund der Größe im Anhang wieder und orientiert sich an einer vereinfachten Version der Referenzschablone aus [PR15, S.74].

2.4.2 Nicht-funktionale Anforderungen

Eine vollständige Beachtung der nicht-funktionalen bzw. Qualitätsanforderungen ist wichtig, da diese maßgeblichen Einfluss auf Kosten und Architektur des Systems haben. [Ebe22, S.83] Des Weiteren ist es nötig, die Randbedingungen für das System zu definieren. Randbedingungen sind Bedingungen, die unseren Lösungsraum die vorgegebenen Anforderungen umzusetzen einschränken. [PR15, S.9] Ein Beispiel wären Richtlinien wie die DSGVO, die regelt, wie wir mit personenbezogenen Daten im System umgehen müssen. Im Folgenden wird ein Überblick über die Qualitätsanforderungen und Randbedingungen des Systems gegeben.

Qualitätsanforderungen

QA/1 Das System soll hochverfügbar sein.

QA/2 Das System soll einfach horizontal skalierbar sein.

QA/3 Das System ermöglicht einen schnellen Release von neuen Versionen ohne Ausfallzeit.

QA/4 Das System soll einfach zu warten sein.

QA/5 Das System soll beobachtbar sein.

Randbedingungen

RB/1 Ein Fremdsystem für den Payment- und E-Mail-Service-Provider wird nicht verwendet, sondern durch Konsolenausgaben im eigenen System selbst simuliert.

RB/2 Das eigentliche Produkt (ZIP-Datei des Templates) wird nicht hochgeladen, da der Verkauf nur simuliert wird.

RB/3 Benutzerkommentare werden als Initialen des Benutzernamens entlang der Spur angezeigt. Wenn der Benutzer mit der Maus über die Initialen fährt, wird eine Box eingeblendet, die den Kommentar anzeigt.

RB/4 Die Initialen und die Benutzerkommentare verhalten sich nicht responsiv zum Browserfenster, die richtige Darstellung ist also nur bei voller Fenstergröße zu gewährleisten.

RB/5 Die Authentifizierung erfolgt durch eine einfache Eingabe von Benutzername und Passwort.

2.5 Problem-Domäne

Nachdem wir den Systemkontext und die Anforderungen des Systems definiert haben, ist es wichtig, strukturelle Softwaremodelle zu erstellen. Sie geben uns einen Überblick über die Komponenten des Systems und deren Beziehung zueinander. [Som18, S.173] Ein Modell, welches uns einen guten Überblick über das Problemfeld der Anforderungen schafft, ist das Domänenklassendiagramm. Dieses UML-Klassendiagramm stellt die wesentlichen Entitäten der Problemdomäne dar, einschließlich ihrer Attribute und Beziehungen zueinander, ohne dabei Methoden oder Verhalten der Klassen zu beschreiben.

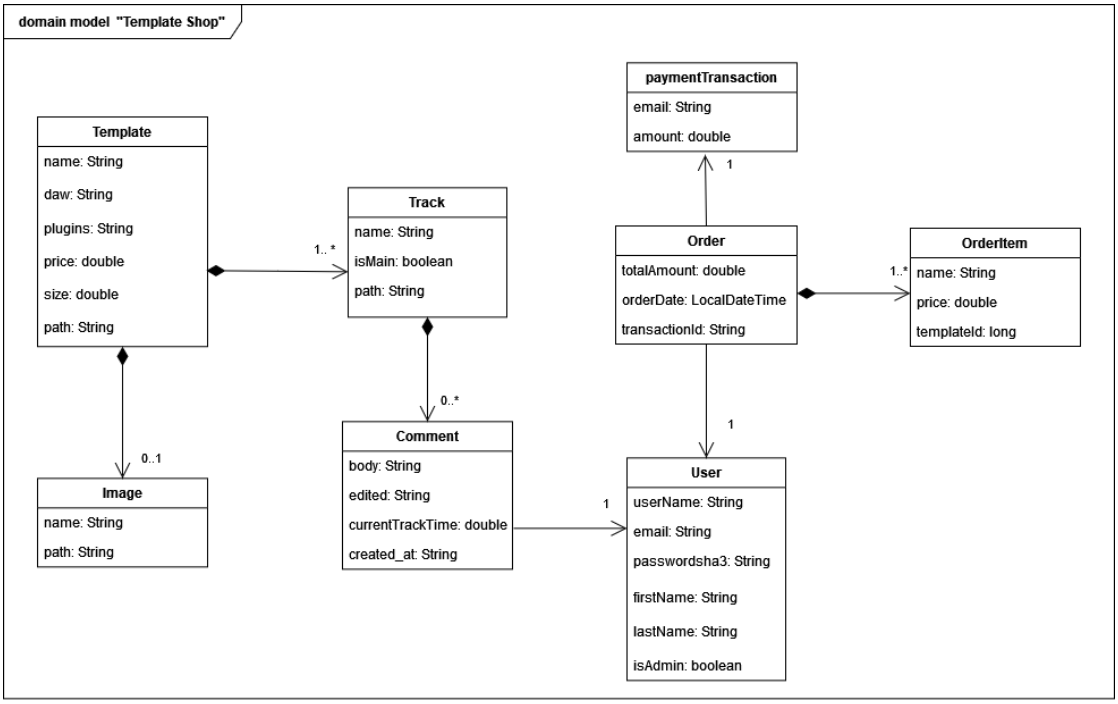

Abbildung 2.3: Domänenklassendiagramm für den Template Shop

Nach Abbildung 2.3 enthält ein Template demnach genau ein Bild (*Image*) und eine oder mehrere Spuren (*Track*). Die Spuren selbst enthalten keine oder beliebig viele Kommentare (*Comment*). Diese Assoziationen sind als Komposition modelliert, da das Löschen eines Templates auch das zugehörige Bild, die Spuren und die Kommentare entfernt.

Im Frontend hat der Nutzer (*User*) die Möglichkeit, Kommentare und Bestellungen (*Order*) zu erstellen. Die Modellierung zeigt jedoch nur die Assoziationen des Kommentars und der Bestellung zu einem Nutzer, da der aktuelle Prototyp keine Funktionen zur Anzeige aller Kommentare und Bestellungen eines Nutzers bietet.

Die Bestellung enthält ein oder mehrere Produkte (*OrderItems*) und verweist auf eine Zahlungstransaktion (*paymentTransaction*). Diese Assoziation wird nicht als Komposition dargestellt, da die Zahlungstransaktion auch nach dem Entfernen der Bestellung im System verbleiben soll. Das Löschen einer Bestellung führt jedoch zum Entfernen der zugehörigen Produkte.

3 Systemdesign

Im folgenden Kapitel wird das System entworfen, welches die zuvor ermittelnden Anforderungen erfüllen soll. Die Dokumentation orientiert sich am arc42-Template, welche eine Vorlage zur Beschreibung und Entwicklung von Software-Architekturen bietet. [SH11, S.47-48] Dabei wird das System aus verschiedenen Sichten betrachtet, welche unterschiedliche Aspekte des Systems sichtbar machen. Die Sichten basieren auf dem „4+1"-Sichtenmodell von Krutchen, welche Grundlage einer vollständigen Entwurfsdokumentation sind. [Som18, S.198-199]

3.1 Verwandte Arbeit

Bevor wir uns auf den spezifischen Systementwurf konzentrieren, sollten wir uns einen Überblick über verwandte wissenschaftliche Arbeiten verschaffen, die ähnliche Probleme behandeln. Auf diese Weise können wir mögliche Architekturansätze für unser System besser evaluieren.

Obwohl es viele wissenschaftliche Arbeiten zu Microservices gibt, sind solche Arbeiten in Bezug auf E-Commerce seltener vorzufinden. Ein geeignetes verwandtes Paper ist dennoch „Microservice Architectures for Scalability, Agility and Reliability in E-Commerce" von OTTO. [HS17] In diesem Paper wird die Umstellung der monolithischen Architektur auf eine Microservice-Architektur für die E-Commerce-Plattform OTTO behandelt. Das Paper ist relevant, da es eine Lösung für ähnliche nicht-funktionale Anforderungen bereitstellt, wie sie auch für unser System definiert wurden. Im Folgenden folgt eine Zusammenfassung des Papers:

Im einleitenden Abschnitt wird die Microservice-Architektur mitsamt wichtigen Aspekten beschrieben: Diese ist ein System aus vielen kollaborierenden Microservices, die jeweils eine Implementierung für einen Geschäftsbereich bieten. Sie zeichnen sich durch lose Kopplung und eventueller Datenkonsistenz aus, um so hohe Verfügbarkeit zu ermöglichen.

Gleichzeitig fördert diese Architektur die Skalierungsmöglichkeiten und Fehlertoleranz der Gesamt-Anwendung, da Fehler nicht auf das gesamte System propagiert werden. Es wird unabhängiges Arbeiten durch cross-funktionale Teams gewährleistet, während die Bereitstellung üblicherweise mithilfe von Containerisierung und Cluster-Management-Infrastrukturen erfolgt.

Im nächsten Abschnitt des Papers wird detailliert auf OTTOs Übergang zur Microservice-Architektur eingegangen:

Die Entscheidung, von einer monolithischen Architektur zu Microservices überzugehen, wurde durch nicht-funktionale Anforderungen wie Skalierbarkeit, Performance und Ausfallsicherheit sowie die Notwendigkeit, Mitarbeiterkapazitäten effizienter zu nutzen, angetrieben. OTTO nutzte Conway's Law, indem mehrere separate Teams an vertikalen Microservices arbeiteten, die jeweils eine Geschäftsdomäne abdeckten. Die Kommunikation zwischen diesen Services erfolgt über einen Backend-Integration-Proxy, der REST APIs verwendet, um hohe Verfügbarkeit sicherzustellen und die Ausbreitung von Fehlern zu verhindern. Durch die Vermeidung gemeinsamer Zustände und Infrastruktur verbesserte sich die horizontale Skalierbarkeit und Ausfallsicherheit des Systems. Kontinuierliche und sichere Deployments wurden durch eine Test- und Delivery-Pipeline gewährleistet. Zudem verfügt jedes Team über ein Dashboard zur Überwachung wichtiger Metriken ihrer Microservices. Abschließend ist eine dynamische Skalierung basierend auf CPU-Auslastung und Arbeitslastschwankungen implementiert, welche optimale Leistung ohne operationelle Eingriffe gewährleistet.

Im letzten Abschnitt wird das Paper dann noch einmal zusammengefasst. Dabei wird betont, dass Microservice-Architekturen trotz ihrer Vorteile mit hohen Kosten verbunden sind: Die Aufrechterhaltung von Konsistenz, Überwachung und Fehlertoleranz in einem verteilten System erfordert eine hohe operationelle Komplexität, die von hoch qualifizierten Entwicklerteams bewältigt werden muss.

Auf Basis der im Paper vorgestellten Lösungsansätze für unsere Anforderungen wird im nächsten Abschnitt ein Architekturstil für unser System ausgewählt.

3.2 Architekturstil

Die Softwarearchitektur legt grundlegende Prinzipien und Strukturen fest, die langfristige Auswirkungen auf die Systemqualität und Systementwicklung haben. [Tre21, S.2-3] Dies

schließt auch die Erfüllung der Qualitätsanforderungen des Systems mit ein. Daher wird zunächst ein Überblick über eine Auswahl an Architekturstilen gegeben und anschließend eine Entscheidung getroffen.

3.2.1 Monolithische Architektur

Ein lang bewährter Ansatz ist die monolithische Architektur. Ein monolithisches System basiert auf einem einzeln entwickelbaren und bereitstellbaren Prozess, was bedeutet, dass alle Komponenten aus denen die Anwendung besteht, als eine Einheit in Betrieb genommen werden. [New21, Kapitel 1.3.1] Die Vorteile dieses Ansatzes sind unter anderem ein vereinfachter Test-, Debug- und Entwicklungsprozess, eine gewährleistete Datenkonsistenz und Integrität innerhalb der Anwendung sowie ein vergleichbar einfaches Deployment. [New21, Kapitel 1.3.5] Während dieser Stil für kleine Anwendungen und Projekte mit den richtigen Anforderungen ausreichend war, brachte er bei größeren und komplexeren Projekten Probleme mit sich: Es ist schwieriger, Abhängigkeiten im Code abzugrenzen, was paralleles Arbeiten an der Codebasis erschwert. Zudem muss selbst bei kleinen Änderungen das gesamte System einen Bereitstellungsprozess durchlaufen, was schnelle Releasezyklen erschwert. Ein weiterer wichtiger Nachteil ist die Skalierung. Benötigt eine bestimmte Funktion der Anwendung mehr Ressourcen, um beispielsweise die Anfragelast abzuarbeiten, so muss dafür die gesamte Anwendung skaliert werden. [Ric18, Kapitel 1.1.3] Dies ist einer der Hauptgründe, wieso beispielsweise Netflix im Laufe seiner Lebenszeit auf eine Microservice-Architektur gewechselt ist. [Fri20a]

3.2.2 Microservice-Architektur

Die Microservice-Architektur wird als eine Applikation definiert, welche auf kleine, unabhängig von einander entwickelbare und bereitstellbare Prozesse basiert. Es handelt sich also um ein verteiltes System, in der die Kommunikation über leichtgewichtige Mechanismen stattfindet, wie z. B. Rest-APIs. [Mar14] Die vorher genannten Nachteile fallen dadurch weg: Durch die starken Modul und Zuständigkeitsgrenzen sind Abhängigkeiten nachvollziehbar, wodurch das parallele Arbeiten in größeren Teams einfacher wird. Schnellere Releasezyklen werden dadurch ermöglicht, dass Deployment-Prozesse sich nur auf einzelne Services beziehen statt der gesamten Anwendung. Einzelne Services können nun skaliert werden, wodurch insgesamt an Ressourcen gespart werden kann. [Ric18, Kapitel 1.5.1] Die Nachteile dieses Stils spiegeln jedoch die Vorteile der monolithischen

Architektur wieder: Als verteiltes System wird Datenintegrität zum Problem, da jeder Service seine eigene Datenbank hat und Transaktionen über mehrere Services stattfinden können. Des Weiteren müssen komplexe Fehlerbehebungsmaßnahmen implementiert werden, da das Netzwerk, über denen die Services miteinander kommunizieren, nicht stabil ist. Die betriebliche Komplexität eines verteilten Systems ist insgesamt erhöht, da nun Technologien eingesetzt müssen, die ein verteiltes System verwalten. [Ric18, Kapitel 1.5.2]

3.2.3 Weitere Architekturansätze

Will man das Beste aus beiden Welten, könnte man einen hybriden Ansatz wählen, indem z. B. einzelne Microservices aus einem Monolithen ausgelagert werden, um so einzelne bestimmte Funktionalitäten kostengünstig zu skalieren. Dieser hybride Ansatz findet häufig dann statt, wenn schrittweise von einer monolithischen Architektur in eine serviceorientierte Architektur migriert wird. [Ric18, Kapitel 13.1.2]

Erwähnenswert ist auch der Modulith, ein Monolith, welcher aber durch sehr gute Codepraktiken möglichst modular aufgebaut ist und anhand seiner Struktur interne Funktionalitäten und Abhängigkeiten klar trennt. [Fri21c]

Zusammenfassend hat jeder Architekturstil seine Vor-und Nachteile, weshalb es wichtig ist, diesen von seinen Anforderungen abhängig zu machen.

3.2.4 Wahl des Architekturstils

Weil wir als Qualitätsanforderungen Hochverfügbarkeit, Skalierbarkeit, Agilität sowie Wartbarkeit definiert haben (**siehe QA/1 - QA/4**) und diese Anforderungen, wie im verwandten Paper gezeigt, besonders gut durch eine Microservice-Architektur erfüllt werden können, fiel die Wahl auf diesen Architekturstil. Wie vorhin erwähnt, ermöglicht diese Architektur die schnelle und kostengünstige Skalierung einzelner Services und bietet durch die Nutzung von Orchestrierungssystemen wie Kubernetes eine hohe Verfügbarkeit. Agilität wird durch die Möglichkeit schneller Releasezyklen einzelner Versionen gewährleistet und vereinfachte Wartbarkeit durch die modulare Natur der einzelnen Services. Die Wichtigkeit der Anforderungen wurde gegenüber der operationellen Komplexität einer Microservice-Architektur vorgezogen.

3.3 Bausteinsicht

Die Bausteinsicht als wichtigste Architektursicht stellt alle zu implementierenden Systembestandteile dar und setzt diese zueinander in Beziehung. Beispiele für Bausteine bzw. Systembestandteile sind Komponenten wie Services oder Datenbanken, aber auch Artefakte und Schnittstellen. [SH11, S.56]

Die Bausteinsicht entscheidet darüber, wie die Services strukturiert werden. Diese Strukturierung kann entweder nach Subdomänen oder nach Unternehmensfunktionen erfolgen. Während sich Letzteres darauf richtet, welche Funktionalitäten das Unternehmen bietet, wie z. B. die Bereitstellung eines Produktkataloges und somit einem Produktkatalogservice, betrachtet man im Ersteren die Subdomänen des Unternehmens nach dem Domain-Driven Design. Dementsprechend würde es eine Subdomäne für das Produktkatalogmanagement geben und somit auch einen Produktkatalogservice. [Ric18, Kapitel 2.2.2-2.2.3] Es wird deutlich, dass beide Ansätze zum gleichen Ergebnis kommen können. Der Unterschied liegt hauptsächlich in der Perspektive, aus der die Services strukturiert werden.

Unser Ansatz in der Bausteinsicht gliedert die Services nach den Unternehmensfunktionen. Jede zentrale Funktionalität wird demnach durch einen eigenen Service abgedeckt, dessen Name klar auf den entsprechenden Funktionsbereich hinweist.

Um die Übersichtlichkeit der Bausteinsicht zu verbessern, wurde sie in drei Teile aufgeteilt.

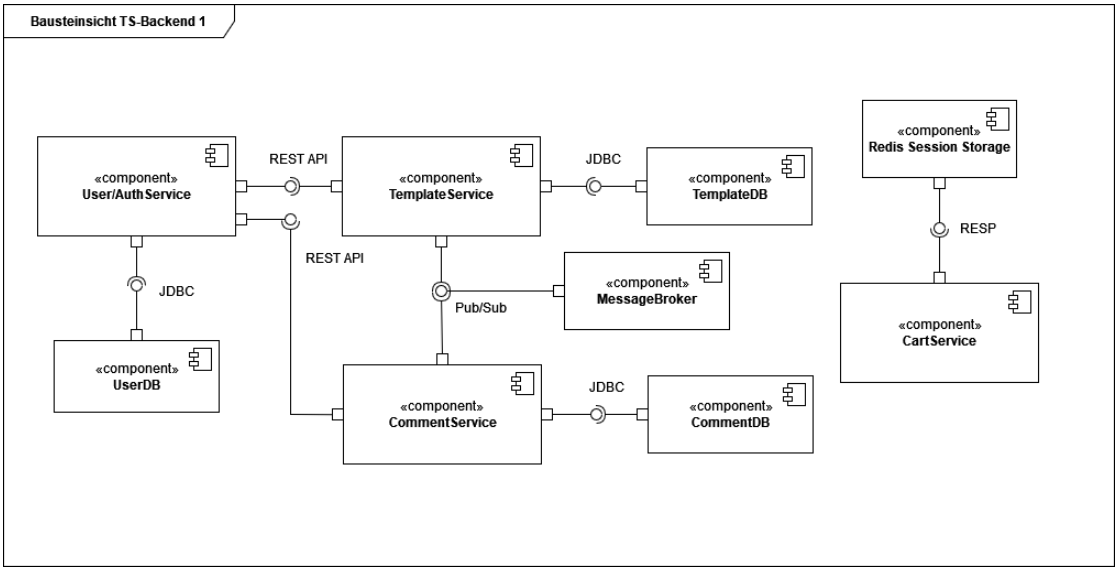

Abbildung 3.1: Bausteinsicht für das Template Shop-Backend 1

Abbildung 3.1 zeigt den ersten Teil des Backends, der die Benutzerverwaltung, Authentifizierung, Kommentarfunktion, Template-Erstellung sowie die Warenkorbfunktionalität abdeckt.

Der Einfachheit halber wurde der *TrackService* mit dem *TemplateService* sowie der *Auth-Service* mit dem *UserService* fusioniert. Dies ist wichtig, da beide Services stark voneinander abhängen und eine Aufteilung unnötig viele Netzwerkanfragen und Datenabhängigkeiten mit sich bringen würde. Da jeder Service auch seine eigene Datenbank hat, werden so autonome, weitgehend entkoppelte und gut skalierbare Services geschaffen.

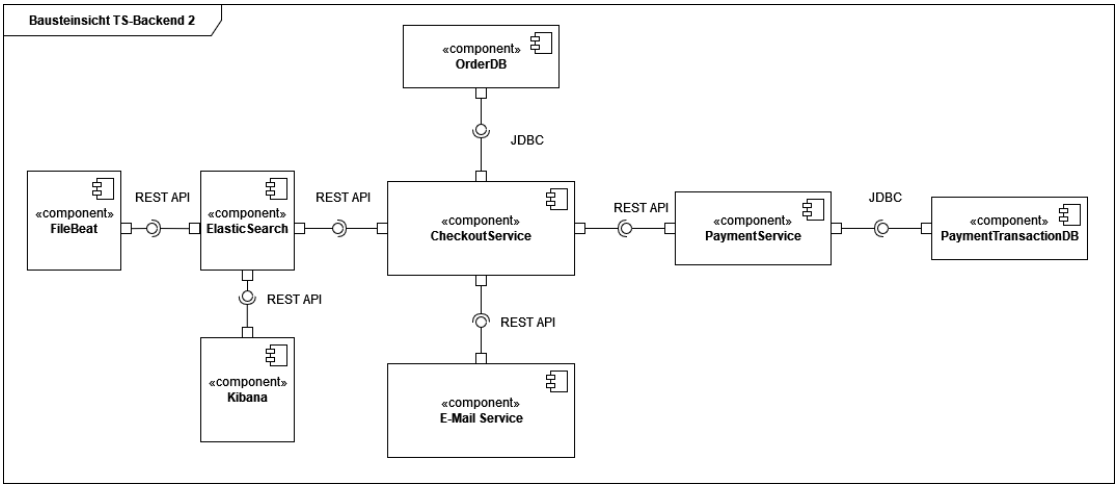

Abbildung 3.2: Bausteinsicht für das Template Shop-Backend 2

Abbildung 3.2 zeigt den zweiten Teil des Backends, der den Monitoring-Stack, den Checkout, die Bezahl- und die E-Mail-Simulation abdeckt.

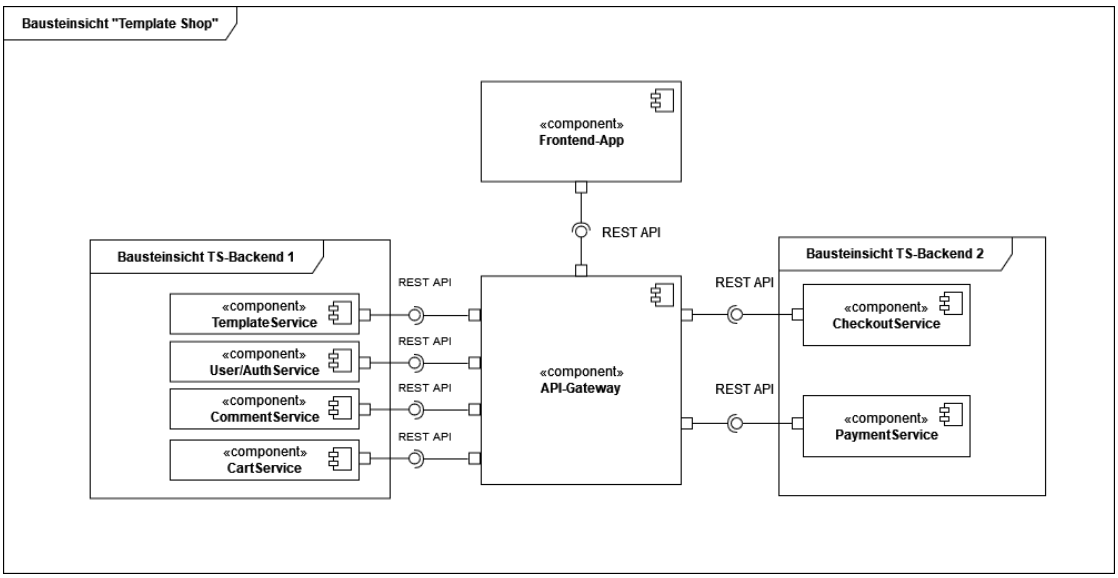

Abbildung 3.3: Bausteinsicht für den Template Shop

Abbildung 3.3 zeigt, wie unsere Frontend-Anwendung über ein API-Gateway mit den genannten Backends bzw. Microservices verbunden ist.

Das API-Gateway dient als zentraler Zugangspunkt für verschiedene Clients und leitet deren Anfragen an die entsprechenden Services weiter. Dies bietet mehrere Vorteile:

Da das Backend für die Clients nur noch eine einzige Adresse bereitstellt, wird der Client-Code von den Backend-Services entkoppelt und dadurch leichter wartbar. Änderungen der Backend-Adressen haben keine Auswirkungen auf die Adresse des API-Gateways. Ähnlich wie beim Facade-Muster leitet das API-Gateway Anfragen von verschiedenen Clients an spezifische APIs weiter, um deren unterschiedlichen Anforderungen gerecht zu werden. Beispielsweise benötigen mobile Clients aufgrund geringerer Netzwerkressourcen angepasste API-Aufrufe. Zuletzt können im API-Gateway weitere Funktionen wie Request-Logging oder Rate-Limiting implementiert werden. [Ric18, Kapitel 8.1.2-8.2.2] Für diese Entscheidung wurden Nachteile wie die erhöhte Komplexität und die leicht verlängerte Anfragedauer durch den zusätzlichen Netzwerk-Hop in Kauf genommen.

3.4 Monitoring

Beim Monitoring geht es darum, den Zustand der laufenden Anwendung zu überwachen. Hierzu gehören beispielsweise Dinge wie die Anzahl der Anfragen pro Sekunde, die Ressourcennutzung und den Zustand der Service-Instanzen. Außerdem ist es wichtig, bei Problemen wie dem Ausfall einer Service-Instanz oder einem sich füllenden Datenträger benachrichtigt zu werden. [Ric18, Kapitel 11.3] Zur Überwachung des Systems gibt es verschiedene Möglichkeiten:

Tracing

Mittels Tracing kann jeder externen Anfrage eine eindeutige ID zugewiesen und ihr Verlauf durch das System von einem Service zum nächsten auf einem zentralen Server aufgezeichnet werden. Ein *Trace* repräsentiert dabei eine externe Anfrage und besteht aus einem oder mehreren *Spans*. Ein *Span* wiederum stellt die Ausführung eines einzelnen Services dar und speichert wichtige Daten wie den Namen, den Startzeitpunkt und den Endzeitpunkt der Operation. Dadurch können detaillierte Einblicke in die Leistung der Services gewonnen werden und mögliche Fehler oder Performance-Engpässe besser nachvollzogen werden. [Ric18, Kapitel 11.3.3] Mögliche Tools zur Implementierung vom Tracing wären Jaeger oder Zipkin.

Metrik-Monitoring

Eine weitere Möglichkeit ist das Sammeln und Überwachen von Anwendungsmetriken. Diese Metriken umfassen sowohl Infrastrukturmetriken wie CPU-, Speicher- und Festplattennutzung als auch anwendungsspezifische Metriken wie die Anzahl der ausgeführten Anfragen. Eine Spring Boot-basierte Anwendung kann beispielsweise durch die Integration der Micrometer Metrics-Bibliothek grundlegende JVM-Metriken sowie anwendungsspezifische Metriken erfassen und an Prometheus übermitteln. Prometheus ist ein Open-Source-Monitoring- und Alarmsystem, welches diese Daten periodisch abfragt und sie zur Visualisierung in Grafana bereitstellt. Es ermöglicht das Einstellen von Benachrichtigungen, wenn selbst definierte Schwellenwerte der Metriken überschritten werden und ermöglicht so eine schnelle Reaktion bei Problemen. [Ric18, Kapitel 11.3.4]

Health-Checks

Bei Health-Checks implementiert ein Service einen Health-Check-Endpunkt, der den Gesundheitszustand der Anwendung zurückgibt. Dies kann beispielsweise durch eine Testabfrage gegen eine Datenbank erfolgen. Die Deployment-Infrastruktur ruft diesen Endpunkt anschließend regelmäßig auf, um den Gesundheitszustand der Instanz zu überprüfen und bei Bedarf entsprechend zu reagieren. [Ric18, Kapitel 11.3.1]

Log-Aggregation

Log-Aggregation ermöglicht die Sammlung und Indexierung aller Logs der Microservices auf einem zentralen Server, der diese analysieren und abfragen lässt. [Ric18, Kapitel 11.3.2]

Wahl der Überwachungsmethode

Da die Überwachung viele Optionen bietet, wurde für den Prototyp zunächst nur die Log-Aggregation verwendet, um die Anwendung einfach zu halten.

Dafür kommt ein EFK-Stack zum Einsatz, welcher aus Filebeat, Elasticsearch und Kibana besteht. Filebeat sammelt hierfür die Logs der Microservices, die über Docker ausgegeben werden und sendet sie an Elasticsearch. Elasticsearch indexiert diese Logs und stellt

eine Suchmaschine bereit, während Kibana sie visualisiert und bei Bedarf in Analyse-Dashboards darstellt. Dadurch kann beispielsweise nach Error-Logs gefiltert werden, um möglichen Problemen auf den Grund zu gehen. Mit der Implementation der Überwachungsmethode erfüllen wir die Anforderung der Beobachtbarkeit. (**siehe QA/5**)

3.5 Kommunikation

Da Microservices sich nicht in einem Prozess befinden, findet Kommunikation üblicherweise über das Netzwerk statt. Hier lassen sich grundsätzlich zwei Kommunikationsarten unterscheiden: die synchrone- und die asynchrone Kommunikation.

3.5.1 Synchrone Kommunikation

Die synchrone Kommunikation arbeitet nach dem Request-Response-Prinzip in der ein Microservice einem anderen Service eine Anfrage stellt und auf die Antwort wartet. [New21, Kapitel 4.3] Mögliche Implementierungen für diesen Kommunikationsstil sind RPC, REST und GraphQL.

RPC

Ein Remote Procedure Call (RPC) ist eine Technik, in der ein lokaler Methodenaufruf einer Anwendung auf einem entfernten Service ausgeführt wird. Dabei wird die Komplexität der Netzwerkkommunikation so abstrahiert, dass sich der RPC wie ein lokaler Methodenaufruf anfühlt. Hierbei werden separate Schema verwendet, welche die Struktur der Nachrichten und Methodenaufrufe definieren, um so eine automatische Generierung von Client- und Server-Stubs für verschiedene Technologien zu ermöglichen. Je nach RPC-Framework kommen verschiedene Serialisierungsmechanismen zur Datenübertragung zum Einsatz, wobei einige Frameworks zusätzlich flexible Optionen bei der Wahl des Netzwerkprotokolls bieten. Trotzdem sollte hier auf erhöhte Kosten, die mit dem Marshaling und Schnittstellenänderungen einhergehen, durch bestimmte RPC-Implementationen geachtet werden. [New21, Kapitel 5.2.1]

REST

REST ist ein architektonischer Stil, der vom Web inspiriert ist und eine Reihe von Prinzipien und Einschränkungen für das Design von Service-Schnittstellen definiert. [New21, Kapitel 5.2.2]

Ein zentrales Konzept von REST ist die Ressource, die typischerweise ein einzelnes Geschäftsobjekt wie einen Kunden oder eine Sammlung solcher Objekte repräsentiert. Diese Ressourcen werden über eine URL referenziert und mithilfe von HTTP-Methoden manipuliert. Beispielsweise können so über GET- und POST-Endpunkte Ressourcen gelesen und erstellt werden, wobei diese häufig in Form von XML-Dokumenten oder JSON-Objekte wiedergegeben werden. Außerdem können mit der Open API Specification Schemata für REST-APIs erstellt werden, die bei Bedarf Client- und Server-Code in verschiedenen Programmiersprachen generieren. Zu den Vorteilen von REST-APIs gehören ihre Einfachheit und die leichte Testbarkeit. Im Gegensatz dazu stehen jedoch die Schwierigkeiten, die sich beim Mapping komplexer Operationen auf einfache HTTP-Verben und der Durchführung von Anfragen über mehrere Ressourcen hinweg ergeben. [Ric18, Kapitel 3.2.1]

GraphQL

Mit GraphQL kann ein Client gezielt spezifische Informationen anfragen, die normalerweise mehrere Anfragen über verschiedene Services hinweg erfordern würden. Man kann GraphQL also als eine Art Aggregations- und Filtermechanismus über Daten von mehreren Services betrachten. Dazu stellt ein Microservice einen GraphQL-Endpunkt bereit, der ein Schema mit allen verfügbaren Datentypen definiert. Clients können dann auf diese mithilfe eines Abfrage-Konfigurators zugreifen. Diese Lösung ist sehr gut für externe Clients wie mobile Geräte und externe APIs geeignet, die viele verschiedene Daten mit einer einzigen Anfrage benötigen. Allerdings sollte GraphQL aufgrund seiner komplexen Caching-Problematik, der ineffizienten Schreibvorgänge und der potenziell hohen Serverlast bei dynamischen Abfragen nicht als einziges Mittel für die Kommunikation zwischen Microservices verwendet werden. [New21, Kapitel 5.2.3]

3.5.2 Asynchrone Kommunikation

Die asynchrone Kommunikation bietet verschiedene Kommunikationsstile an und zeichnet sich dadurch aus, dass der Client nach der Anfrage nicht auf eine sofortige Antwort eines Services warten muss. Neben dem asynchronen Request-Response-Prinzip kann die Kommunikation auch über ein Publish/Subscribe-Modell (im Folgenden pub/sub-Modell) erfolgen, bei dem der Client eine Nachricht veröffentlicht, die von interessierten Services konsumiert wird. [Ric18, Kapitel 3.1.1]

Brokerbasierte Architektur

Eine Möglichkeit asynchrone Kommunikation umzusetzen, ist die brokerbasierte Architektur. In der brokerbasierten Architektur senden alle Services ihre Nachrichten an einen zentralen Nachrichtenbroker, welcher diese wiederum an die jeweiligen Empfänger verteilt. Einerseits ermöglicht der Einsatz eines Brokers eine lose Kopplung zwischen Sender und Empfänger, da der Sender keine spezifischen Informationen über die Empfänger benötigt. Andererseits sorgt der Nachrichtenpuffer des Brokers für hohe Verfügbarkeit, indem er es ermöglicht, dass ein Onlineshop Bestellungen auch dann annehmen kann, wenn das Bestellsystem vorübergehend nicht erreichbar ist. Die Nachteile dieses Ansatzes betreffen potenzielle Performance-Engpässe und die Gefahr eines Single Points of Failure durch den Broker. Moderne Broker-Lösungen sind jedoch auf Hochverfügbarkeit und Skalierbarkeit optimiert, um diese Probleme zu umgehen. Trotzdem bleibt der erhöhte operationale Aufwand für die Installation und Konfiguration des Brokers bestehen. [Ric18, Kapitel 3.3.4]

Brokerlose Architektur

In einer brokerlosen Architektur kommunizieren Services direkt miteinander, ohne einen Nachrichtenbroker zu nutzen. Ein bekanntes Beispiel hierfür ist ZeroMQ, das verschiedene Übertragungsarten wie TCP und UNIX-Sockets anbietet. Der Vorteil dieser Architektur liegt in der vereinfachten Systemstruktur mit verbesserter Latenz und reduziertem Netzwerkverkehr. Dennoch können wichtige Funktionen wie Service-Discovery und eine garantierte Nachrichtenzustellung schwieriger zu realisieren sein, weshalb viele Unternehmensanwendungen auf eine brokerbasierte Architektur setzen. [Ric18, Kapitel 3.3.4]

3.5.3 Wahl der Kommunikationsform

Im Gegensatz zur synchronen Kommunikation verbessert asynchrone Kommunikation die Hochverfügbarkeit des Systems. [Ric18, Kapitel 3.4] Anhand unserer Anforderung der Hochverfügbarkeit (**siehe QA/1**) wäre es naheliegend, jegliche Kommunikation unserer Microservices als asynchrone Kommunikationsform zu implementieren.

Jedoch findet in den Backend-Microservices aufgrund ihres Schnitts nur minimale Kommunikation statt. Die meisten Anfragen bestehen darin, einen Schlüssel zur Verifizierung der Signatur eines JSON-Web-Tokens zu erhalten. Da diese Anfragen einfache GET-Operationen sind, die keinen langen Verifikationsprozess erfordern, wurde beschlossen, hierfür doch eine synchrone REST-API zu verwenden. Alternativen wie GraphQL oder RPC bieten für diesen einfachen Anwendungsfall keinen nennenswerten Mehrwert. Stattdessen wird die Einfachheit der Implementierung einer synchronen API der potenziellen Komplexität und den minimalen Verfügbarkeitsgewinnen einer asynchronen Kommunikationsform vorgezogen.

Die Kommunikation zwischen dem Template- und dem Comment-Service findet asynchron über einen Nachrichtenbroker statt, da der Template-Service keine direkte Rückmeldung vom Comment-Service benötigt. Dieser wird lediglich beim Löschen eines Templates beauftragt, alle zugehörigen Kommentare zu löschen. Zudem soll hier der Nachrichtenbroker als neue Technologie erprobt werden, um diesen eventuell für zukünftige Versionen zu nutzen, insbesondere wenn die Kommunikation zwischen den Microservices komplexer wird. Die Entscheidung für den Nachrichtenbroker basiert auf den Vorteilen der Hochverfügbarkeit, der garantierten Nachrichtenübermittlung und der damit verbundenen Verbesserung der Datenkonsistenz.

Für die Implementierung des Nachrichtenbrokers eignet sich Apache Kafka. Kafka ist ein verteiltes System für Event-Streaming, das Nachrichtenvermittlung nach dem pub/sub-Modell bereitstellt. Durch die Möglichkeit Topics zu partitionieren und replizieren, bietet Kafka hohe Verfügbarkeit und Fehlertoleranz [Kaf]. Mit dem pub/sub-Modell können so Nachrichten vom Template-Service an ein spezifisches Topic veröffentlicht werden, welches der Comment-Service wiederum abonnieren kann. Die einfache Integration in das später verwendete Framework Spring Boot durch die Unterstützung von Spring Kafka war ein Hauptgrund für die Auswahl dieser Technologie.

3.6 Verteilte Transaktionen und Sagas

In einer Microservice-Architektur sind verteilte Transaktionen entscheidend, um Datenkonsistenz über mehrere Services hinweg sicherzustellen. Dies ist notwendig, wenn Operationen der Gesamtanwendung verschiedene Services beinhalten, die jeweils ihre eigenen Datenbanken haben. Das liegt daran, dass jeder Service lokal mit ACID-Transaktionen arbeitet und von allen anderen Services isoliert ist. Wenn jetzt eine Operation mehrere Services betrifft und einer dieser Services ausfällt, würde dies ohne eine verteilte Transaktion zu Dateninkonsistenzen führen. [Ric18, Kapitel 4.1]

Leider ist der traditionelle Ansatz von verteilten Transaktionen wie das Zwei-Phasen-Commit-Protokoll (2PC) nicht für alle moderne Anwendungen geeignet, da Technologien wie NoSQL-Datenbanken oder Message-Broker hier keine Unterstützung anbieten. Zudem führen verteilte Transaktionen zu einer Verringerung der Verfügbarkeit der Gesamtanwendung, da alle beteiligten Dienste gleichzeitig verfügbar sein müssen, um die Transaktion abzuschließen. Um diese Problematik zu lösen, kommt das Saga-Muster zum Einsatz: Sagas arbeiten mit einer Sequenz von lokalen Transaktionen, die durch asynchrone Nachrichten koordiniert werden, um Datenkonsistenz zu gewährleisten. Jeder Abschluss einer lokalen Transaktion im Service löst dabei die Ausführung der nächsten aus. Ein wichtiger Vorteil der asynchronen Nachrichtenübermittlung ist hier, dass alle Schritte einer Saga ausgeführt werden, selbst wenn einige Teilnehmer vorübergehend nicht verfügbar sind. Sollte hierbei in einer Sequenz von lokalen Transaktionen ein Fehler auftreten, werden alle Änderungen der Transaktionen durch Kompensationstransaktionen rückgängig gemacht. [Ric18, Kapitel 4.1.2-4.1.3]

Die Implementierung des Saga-Musters kann entweder durch Choreografie oder Orchestrierung erfolgen. Bei der Choreografie tauschen die Teilnehmer der Transaktionssequenz asynchron Nachrichten aus, um den Ablauf der Saga zu koordinieren. Im Gegensatz dazu übernimmt bei der Orchestrierung ein zentraler Orchestrator die Koordination. Dieser Orchestrator überwacht den Fortschritt der Saga und stellt sicher, dass alle notwendigen Schritte ausgeführt werden. Der Orchestrierungsansatz ist verständlicher und eignet sich besser für komplexere Sequenzen. Es ist jedoch wichtig, darauf zu achten, nicht zu viel Logik in den Orchestrator zu integrieren. Der Choreografie-Ansatz sollte aufgrund seiner schwer nachvollziehbaren Implementierung und möglichen Zyklusabhängigkeiten nur bei sehr einfachen Sequenzen angewendet werden. Jedoch ist wichtig zu erwähnen, dass in verteilten Systemen, die auf Sagas basieren, Anomalien auftreten können, die sich aus

dem Fehlen der Isolationseigenschaft von ACID-Transaktionen ergeben. Isolation verhindert nämlich, dass bei gleichzeitiger Ausführung mehrerer Transaktionen diese sich gegenseitig beeinflussen und dadurch inkonsistente oder unvorhersehbare Ergebnisse entstehen. Bei Sagas sind jedoch Aktualisierungen untereinander sofort verfügbar. Deshalb müssen hier zusätzliche Gegenmaßnahmen implementiert werden, um Anomalien wie Dirty Reads oder verlorene Updates zu verhindern, was zusätzlichen Aufwand erfordert. [Ric18, Kapitel 4.2-4.3]

Wir sehen, dass verteilte Transaktionen und Sagas die Komplexität unserer Implementation stark erhöhen können. Deshalb wurden die Services so strukturiert, dass keine Transaktionen über mehrere Services stattfinden. Lediglich ein Fehler beim Löschen eines Templates könnte dafür sorgen, dass Kommentare für ein gelöschtes Template existieren. Die Auswirkungen dieser Dateninkonsistenz wären minimal und würden nur bei einer sehr großen Anzahl fehlgeschlagener Löschaktionen zu Speicher- und Performanceeinbußen in der Datenbank führen. In der Praxis kommen solche Löschaktionen jedoch nur vereinzelt vor, da ein Verkäufer selten ein erstelltes und erfolgreich verkauftes Template löscht.

3.7 Datenmodell

Jeder Service führt eine relationale Datenbank für seine Entitäten. Aufgrund der einfachen, nicht verschachtelten Struktur der Daten und ausgereiften Implementierung dieses Modells von verschiedensten Anbietern und Frameworks wurde das relationale Datenmodell ausgewählt. Da die Wahl des Datenbankmanagementsystems keinen speziellen Anforderungen unterliegt, fällt diese auf PostgresSQL. Unterstützt wird sie durch dessen Modernität und Unterstützung im später erwähnten Google Cloud SQL.

3.8 Warenkorb

Um einen Warenkorb zu realisieren, ist es sinnvoll, dem Client eine Session mit Warenkorbinformationen zuzuteilen. Dafür gibt es unter anderem zwei Lösungsansätze:

In der ersten Variante speichert man die Daten des Warenkorbs in einem Cookie im Client und spart sich dadurch die Verwaltung einer Datenbank, langsame Datenbankzugriffe auf den Warenkorb sowie eine erschwerte Skalierbarkeit der Warenkorbfunktion. Das

Problem hier wäre der zusätzliche Overhead, der durch ein größeres Cookie entsteht und womöglich die Anfragen verlangsamt. Auch wären bei schlechter Implementierung erhöhte Sicherheitsbedenken wie das Risiko der Datenmanipulation oder das potenzielle Offenlegen sensibler Informationen möglich.

In der zweiten Variante speichert man die benötigten Daten in einer Datenbank. Damit können Warenkorb-Informationen über verschiedene Geräte des Clients gespeichert und evtl. zusätzliche Funktionalitäten des Warenkorbs gestaltet werden. Die oben genannten Nachteile wie langsame Datenbankzugriffe als auch erschwerte Skalierbarkeit gehen aber auch einher. Üblicherweise ist eine längere Persistenz bei solch temporären Daten auch nicht notwendig.

Ein gutes Mittelmaß ist die Implementierung einer Warenkorbfunktionalität mittels eines Key Value Store als Datenbank in Kombination mit einem Session-Cookie. Dabei würden die Produkte im Warenkorb im Key Value Store gespeichert werden und auf eine Session-ID im Cookie verwiesen werden. Key Value Stores im Hauptspeicher bieten eine hohe Geschwindigkeit der Datenverarbeitung und durch ihre Natur eine hohe Skalierung mittels Sharding. [KM23, S.254] Als Key Value Store Implementation würde auf Redis gesetzt werden, da es solche sehr schnellen Lese- sowie Schreiboperationen bietet und durch das unterstützte Clustering hochskalierbar ist. Ein weiterer Grund ist, dass Redis in Verbindung mit Spring Session eine ausgereifte Implementierung in unserem geplanten Backend-Framework vorfindet. Durch diese Kombination wäre eine performante und skalierbare Lösung geboten, die bei Bedarf auch persistiert werden kann, umso die Vorteile beider Lösungsansätze zu vereinen.

3.9 Sicherheit

Auch wenn Sicherheit in dieser Prototyp-Version keine zentrale Rolle spielt, wurden dennoch einige Sicherheitsmaßnahmen implementiert. Für die Authentifizierung, also der Frage, ob der Nutzer der ist, für den er sich ausgibt, wird eine simple Kombination aus Passwort und Nutzernamen angelegt. (**siehe RB/5**) Beim Registrierungsprozess wählt der Nutzer sein Passwort aus, welches dann in der Datenbank des *UserService* als Hash gespeichert wird. Für die Autorisierung, also ob der Nutzer Zugriff auf geschützte Ressourcen hat, wurde Folgendes bestimmt:

Bei der Registrierung eines Nutzers erstellt der *UserService* einen JSON-Web-Token (JWT), der mit einem asymmetrischen Schlüssel signiert wird. Ein JWT ist ein Base64 codiertes und signiertes JSON, welches alle relevanten Informationen eines Benutzers, die für eine Authentifizierung oder Autorisierung nötig sind, enthält. Wenn ein Service nun überprüfen muss, ob ein Nutzer für die benötigte Ressource autorisiert ist, verifiziert er die Signatur des mitgeschickten JWT vom Client anhand des öffentlichen Schlüssels des *UserService*. Aufgrund der Wahl einer asymmetrischen Verschlüsselung müssen die Microservices keinen Schlüssel untereinander aushandeln, was bei einem verteilten System mit vielen Services von Vorteil ist. Auch wird durch den Verzicht der Speicherung von Zustand sowie die Eliminierung des Bedarfs an Datenbanken für Authentifizierungs- und Autorisierungsinformationen die Skalierbarkeit der Microservice verbessert und die Ressourcennutzung optimiert. Nachteile wie der erhöhte Overhead eines Tokens oder die verringerte Sicherheit bei Komprimierung des Tokens wurden aufgrund der genannten Vorteile in Kauf genommen.

Ähnliche Sicherheitsvorkehrungen wurden auch beim *PaymentService* getroffen: Dieser generiert für eine Transaktion ebenfalls einen JWT, damit der *CheckoutService* überprüfen kann, ob die Transaktion tatsächlich stattgefunden hat und nicht gefälscht wurde. Somit haben *Guests* Zugriff auf alle öffentlich zugänglichen APIs, jedoch ohne Kundenkonto nicht auf geschützte.

3.10 Laufzeitsicht

Die Laufzeitsicht zeigt, wie die einzelnen Bausteine des Systems zur Laufzeit zusammenarbeiten. [SH11, S.66] So werden die verschiedenen Funktionen und Kommunikationsabläufe der Komponenten zu bestimmten Anwendungsfällen deutlich. Im Folgenden werden die kommunikationsrelevantesten Use Cases mit UML-Sequenzdiagrammen dargestellt.

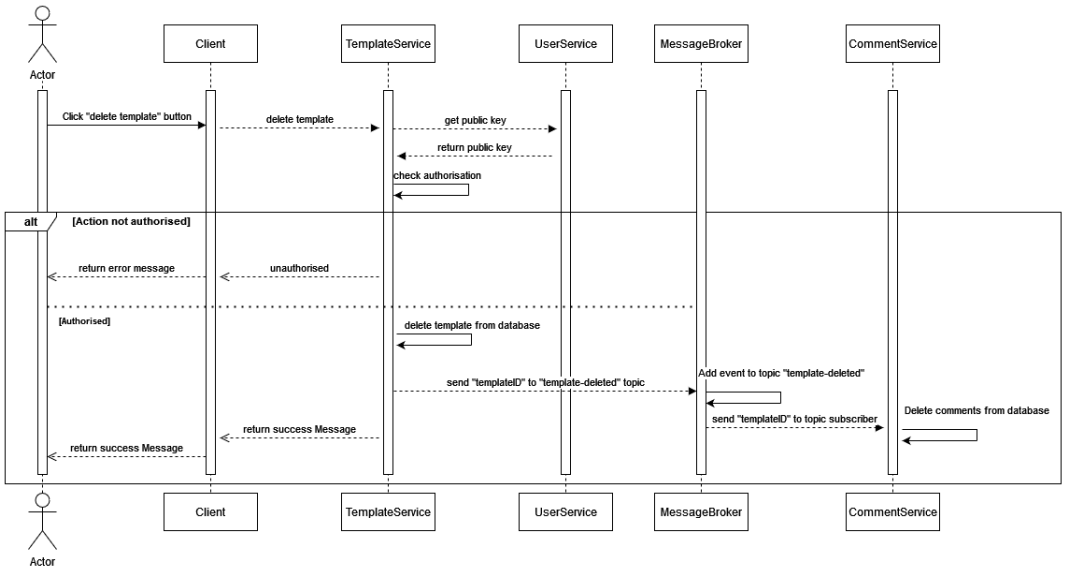

Abbildung 3.4: UML Sequenzdiagramm für das Löschen eines Templates (uc/15)

Abbildung 3.4 zeigt den Ablauf des Löschens eines Templates zur Laufzeit. Vor dem Fortfahren des Löschvorgangs fordert der *TemplateService* den öffentlichen Schlüssel (*public key*) vom *UserService* an, um die Autorisierung der Aktion zu überprüfen. Der *TemplateService* sendet anschließend ein Event an den *MessageBroker*, um mit dem *CommentService* zu kommunizieren und setzt seinen Ablauf fort. Der *MessageBroker* leitet das Event an den *CommentService* weiter, wodurch alle Kommentare zu dem Template gelöscht werden, ohne dass der *TemplateService* blockiert wird.

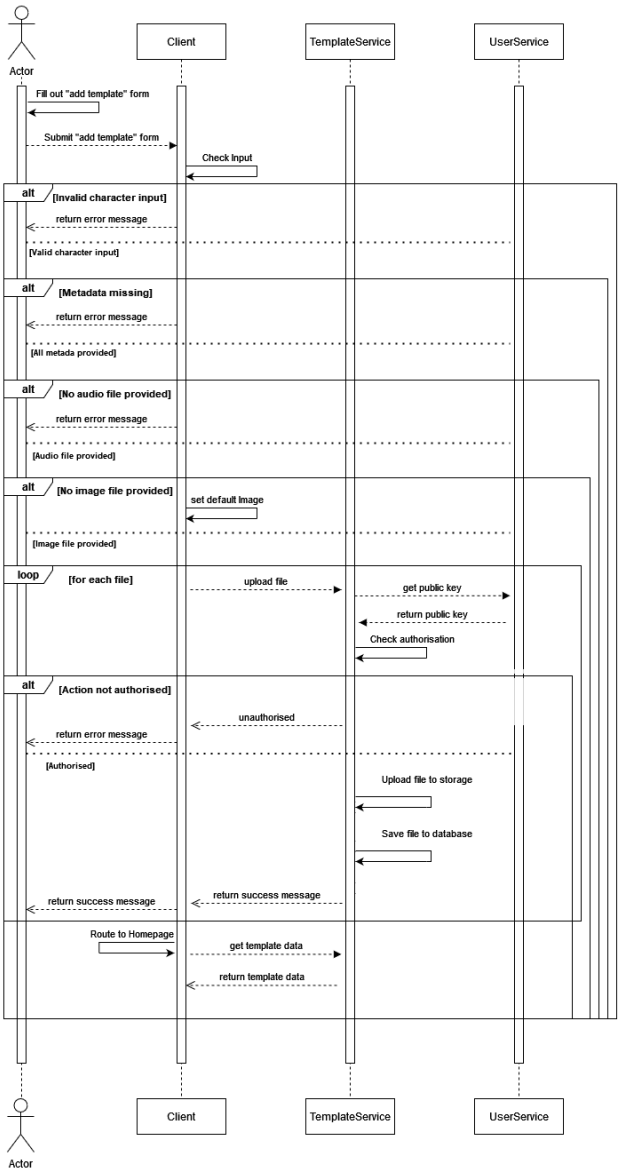

Abbildung 3.5: UML Sequenzdiagramm für das Hinzufügen eines Templates zum Shop (uc/17)

In Abbildung 3.5 wird der Prozess des Hinzufügens eines Templates zur Laufzeit dargestellt. Der Client überprüft zunächst alle Eingaben des Akteurs und stellt sicher, dass eine Audiodatei bereitgestellt wurde. Falls kein Bild ausgewählt wurde, verwendet der Client ein Standardbild. Anschließend sendet der Client jede Datei an den *TemplateService*, der die Autorisierung über den *UserService* überprüft. Bei erfolgreicher Autorisierung wer-

den die Dateien auf der Festplatte und in der Datenbank gespeichert. Der Client leitet den Akteur zur Startseite weiter und aktualisiert die Template-Dateien.

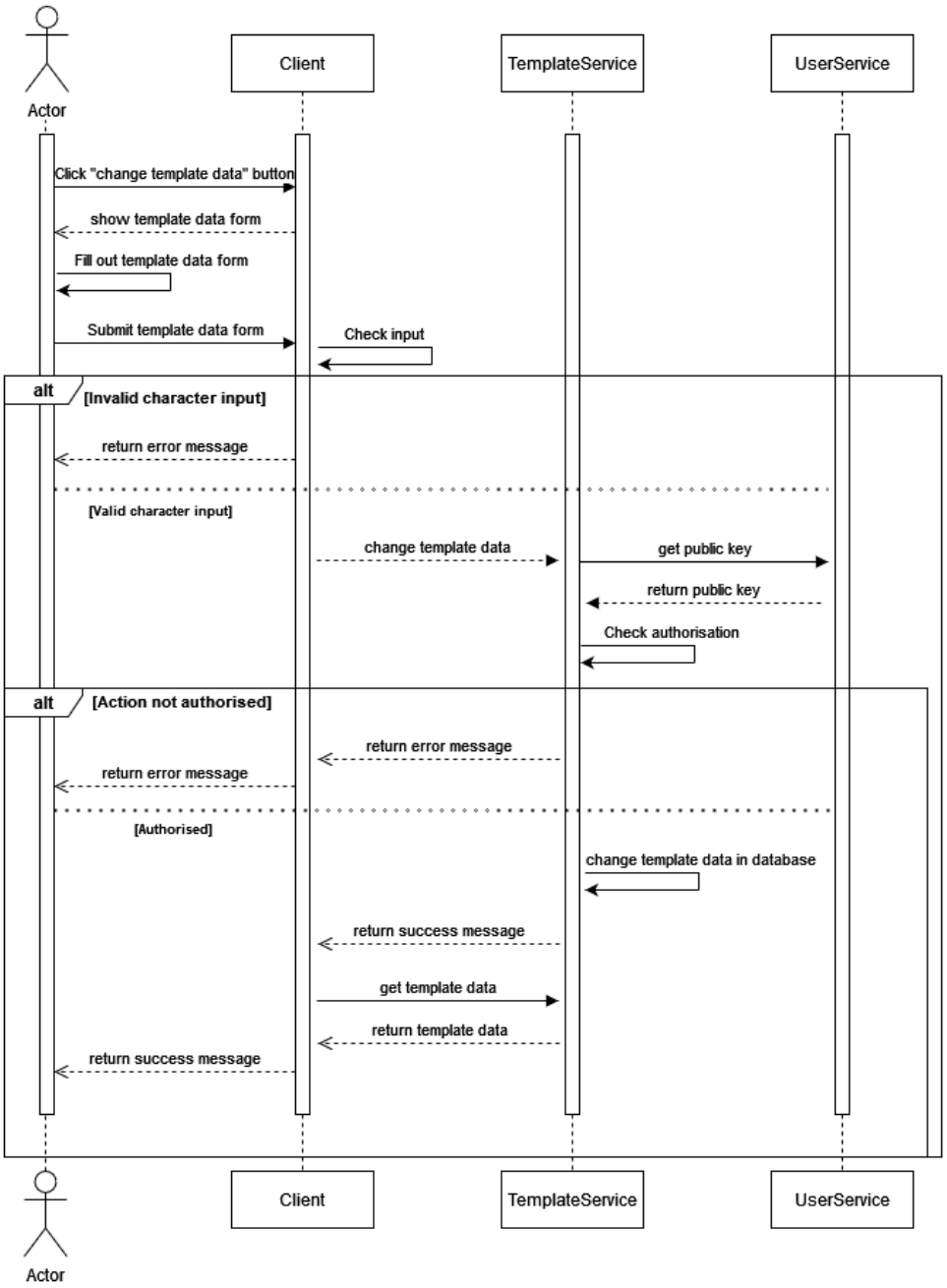

Abbildung 3.6: UML Sequenzdiagramm für das Bearbeiten eines Templates (uc/16)

Abbildung 3.6 veranschaulicht den Prozess des Bearbeitens eines Templates zur Laufzeit. Der Ablauf ähnelt dabei dem Upload-Vorgang: Nachdem der Akteur ein Formular zum Ändern der Template-Metadaten ausgefüllt hat, überprüft der Client die Eingaben auf mögliche Fehler. Die aktualisierten Metadaten werden anschließend an den *TemplateService* gesendet, der zur Autorisierung erneut den *UserService* anfragt. Nach erfolgreicher Autorisierung werden die Änderungen in der Datenbank aufgenommen. Der Client lädt die aktualisierten Template-Metadaten nach und zeigt dem Akteur eine Erfolgsmeldung an.

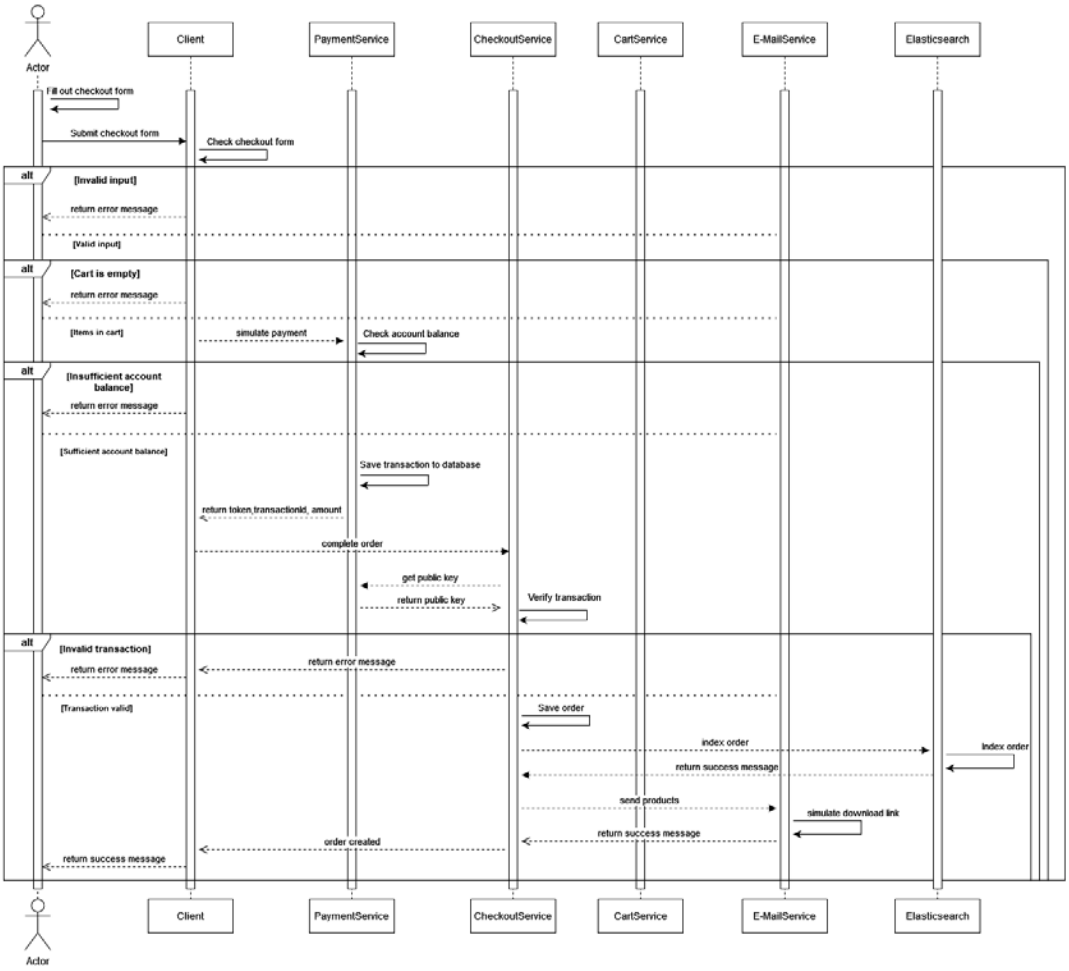

Abbildung 3.7: UML Sequenzdiagramm für das Kaufen eines Templates (uc/9)

Abbildung 3.7 stellt den Prozess des Template-Kaufs zur Laufzeit dar. Nachdem der Akteur das ausgefüllte Checkout-Formular einsendet, überprüft der Client die Daten auf Fehler und stellt sicher, dass Produkte im Warenkorb vorhanden sind. Nach erfolgreicher

Überprüfung leitet der Client eine Anfrage an den *PaymentService* weiter, der simulativ die Deckung des Kontos überprüft. Der *PaymentService* speichert die Transaktion und sendet einen Token mit den Transaktionsdaten an den Client zurück. Dieser leitet die Daten an den *CheckoutService* weiter, der den öffentlichen Schlüssel (*public key*) des *PaymentService* anfordert, um die Transaktion zu verifizieren. Nach der erfolgreichen Verifizierung speichert der *CheckoutService* die Bestellung und indexiert sie in Elasticsearch. Der *CheckoutService* informiert den *E-MailService*, um die simulierten Downloadlinks zu erstellen. Nach der Generierung der Links benachrichtigt der *CheckoutService* den Client über den erfolgreichen Kauf, welcher wiederum eine Erfolgsmeldung anzeigt.

3.11 Deployment

Als Deployment wird die Inbetriebnahme und Wartung einer Software bezeichnet, wobei früher Entwicklung und Betrieb strikt voneinander getrennt waren. Heute hingegen hat sich zunehmend eine Kultur und ein Entwicklungsprozess etabliert, der beide Bereiche miteinander verbindet. Diesen Wandel und die darausfolgende neue Methodik beschreibt der Begriff Devops, welcher unter anderem Prinzipien und Entwicklungspraktiken definiert, die das Deployment einer Anwendung beschleunigen und automatisierbar machen. [IBM] Dazu zählen z. B. Continuous-Integration- und Continuous-Delivery-Pipelines, welche bei Codeänderungen automatische Test-, Build- und Deployment-Prozesse starten, die schnell eine sichere und neue Version der Software bereitstellen sollen. Dies bedeutet aber auch, dass Entwickler sich mehr Wissen und Fähigkeiten aneignen müssen, um Devops erfolgreich zu implementieren.

Im Folgenden werden die Grundlagen für das Deployment einer skalierbaren Microservice-Architektur geschaffen und das System aus einer Deployment- bzw. Verteilungssicht betrachtet.

3.11.1 Bereitstellungsoptionen

Um eine Anwendung in Betrieb zu nehmen, kann sie als sprachspezifisches Paket, innerhalb einer VM oder als Container bereitgestellt werden:

Bereitstellung als sprachspezifisches Paket

Bei dieser Bereitstellungsoption wird die Anwendung in einem Format bereitgestellt, das von der verwendeten Programmiersprache abhängt. Ein in Java geschriebener Service wird beispielsweise als ausführbare JAR- oder WAR-Datei bereitgestellt. Ein Nachteil dieser Herangehensweise ist, dass detaillierte Kenntnisse über die spezifische Programmierumgebung erforderlich sind, um alle Anforderungen für den Betrieb der Anwendung korrekt zu installieren. Zudem kann die Ressourcennutzung nicht eingeschränkt werden, was aufgrund der fehlenden Isolation der Prozesse dazu führen kann, dass ein ressourcenintensiver Prozess andere Prozesse negativ beeinträchtigt. Eine komplexe automatische Verteilung der Instanzen auf Maschinen, wie sie von Orchestrierungsframeworks angeboten wird, muss manuell konfiguriert werden. [Ric18, Kapitel 12.1]

Bereitstellung als virtuelle Maschine

Bei dieser Bereitstellungsmethode wird die Anwendung von einer Deployment-Pipeline in ein virtuelles Maschinen-Image verpackt und in einer Produktionsumgebung bereitgestellt. Dies kann beispielsweise mit Amazon Machine Images (AMIs) auf AWS EC2 Instanzen oder ähnlichen Technologien in Cloud-Umgebungen umgesetzt werden. Vorteilhaft ist, dass das VM-Image alle Softwareabhängigkeiten zur Ausführung des Services enthält, wodurch die Einrichtung einer separaten Umgebung wegfällt. Die Anwendung kann dabei auf jedem System, das VM-Images unterstützt, bereitgestellt werden. Zudem wird jede Service-Instanz in der VM isoliert ausgeführt, wodurch der Einfluss auf Ressourcen durch andere Service-Instanzen ausbleibt. Funktionen wie Lastenverteilung und Skalierung werden hier durch moderne Cloud-Infrastrukturen anwendbar. Ein Nachteil der virtuellen Maschinen ist der zusätzliche Overhead, den ein gesamtes Betriebssystem mit sich bringt, welches zu einer weniger effizienten Ressourcennutzung und langsameren Bereitstellungszeiten führt. Um diese Nachteile zu umgehen, bietet es sich an, Services als Container bereitzustellen. [Ric18, Kapitel 12.2]

Bereitstellung als Container

Ein Container führt die Anwendung ähnlich wie eine virtuelle Maschine in einer isolierten Umgebung aus. Da die Virtualisierung jedoch auf Betriebssystem-Ebene stattfindet, werden alle Container trotzdem auf demselben Host-System ausgeführt. Um die Isolation

zusätzlich zu verbessern und Portkonflikte zu vermeiden, wird jedem Container eine eigene IP-Adresse und ein eigenes Dateisystem zugeteilt. Beim Bereitstellungsprozess wird dann zuerst ein Container-Image erstellt und in einer Registry gespeichert. Ein Image selbst ist ein Bauplan einer Anwendung, welcher alle Abhängigkeiten, die diese Anwendung benötigt, enthält. Bei Ausführung wird dieses Image nun heruntergeladen und zur Erstellung des Containers verwendet. Ein entscheidender Vorteil gegenüber der Bereitstellung mit VMs ist die geringe Größe der Container. Durch das fehlende Betriebssystem können Container Images so viel schneller gebaut und bereitgestellt werden. Sofern keine Cloud Lösung benutzt wird, die die zugrunde liegende Infrastruktur betreut, muss diese jedoch selbst verwaltet werden. [Ric18, Kapitel 12.3]

Wahl der Bereitstellung

Die Anwendung wird mittels Containern in Betrieb genommen, weil sie aufgrund ihrer Isolation, Kapselung und geringen Overhead die schnellsten und zuverlässigsten Deployments bieten. Wegen seiner Vertrautheit und Beliebtheit wurde Docker zur Containerisierung verwendet.

3.11.2 Docker

Docker ist eine Plattform zur Containerisierung von Anwendungen. Mit Dockerfiles können Images und deren Abhängigkeiten textuell definiert werden, wobei das selbstdefinierte Image aus einer Reihe von vordefinierten Images wie Linux Distributionen aufgebaut werden kann. Ein oder mehrere Container, welche die Anwendung bilden, werden mithilfe einer Container-Runtime ausgeführt. Wichtig anzumerken ist, dass Container selbst keinen persistenten Datenspeicher anbieten. Dieser muss über beispielsweise Docker-Volumes selbst eingerichtet werden.

Um nicht alle Container einzeln ausführen zu müssen, kann eine docker-compose Datei definiert werden. In dieser YAML-Datei können jegliche Container und deren Konfiguration spezifiziert werden. Beim Ausführen dieser Datei erstellt Docker ein Netzwerk, in dem alle konfigurierten Container laufen und sich unter ihrem Containernamen finden können. Zwar ist dieser Ansatz ausreichend, um eine Container-Anwendung lokal zu testen, reicht jedoch alleine nicht aus, um einen sicheren Produktionsbetrieb zu gewährleisten. Hierfür wird ein Container-Orchestrierungs-Tool benötigt, das die Container

dauerhaft überwacht, bei Bedarf schnell repliziert und automatisch skaliert. Ein Beispiel für ein solches Tool ist Kubernetes:

3.11.3 Kubernetes

Ursprünglich von Google entwickelt und als Open Source freigegeben, bietet Kubernetes eine Container-Orchestrierungsplattform, welche die Bereitstellung, Skalierung und Verwaltung von Containern automatisiert. Dies geschieht durch Konfigurationsdateien, welche Soll-Zustände der Container definieren.

Durch diese Automatisierung kann Hochverfügbarkeit in verteilten Systemen erreicht werden, da durch Kubernetes eine Lastenverteilung auf verschiedene virtuelle Server stattfindet, Container bei Bedarf automatisch repliziert werden und abgestürzte Container neu hochgefahren werden können. [Ric18, Kapitel 12.4] Zum besseren Verständnis von Kubernetes und der Verteilungssicht des Systems folgt eine Erklärung der Architektur und schließlich der Komponenten dieser Orchestrierungsplattform:

Kubernetes unterscheidet zwischen zwei Servertypen, die Teil eines Kubernetes-Clusters sind: Den Master und den Worker Node. Der **Master Node** steuert und verwaltet das gesamte Cluster, indem er Konfigurations- und Zustandsdaten speichert, die API bereitstellt und für die Bereitstellung neuer Container sorgt. [Wel24, S. 46-47] Um dies zu bewerkstelligen, laufen auf ihm verschiedene Prozesse:
Der **kube-apiserver** ist eine zentrale Komponente im Kubernetes-Cluster, die alle internen und externen API-Anfragen verarbeitet. Er ist verantwortlich für die Validierung und Autorisierung von Anfragen, das Überwachen von Rate Limits und Quotas sowie die Verbindung zur etcd-Datenbank. [Wel24, S. 48]

Die **etcd-Datenbank** speichert alle Konfigurationsdaten des Clusters in hochverfügbarer und konsistenter Weise. [Wel24, S. 49]

Der **kube-controller-manager** kontrolliert per Heartbeart die Gesundheit der Nodes, die korrekte Anzahl laufender Container, die Erstellung von Serviceaccounts sowie API-Tokens und stellt die Verbindung zwischen Containern und Services her. [Wel24, S. 49-51]

Der **kube-scheduler** ist verantwortlich für die Zuweisung von Pods zu Nodes im Kubernetes-Cluster, indem er die verfügbaren Ressourcen wie CPU und Speicher sowie die festgelegten Scheduling-Regeln berücksichtigt. [Wel24, S. 48]

Der **cloud-controller-manager** ist für die Kommunikation zwischen Kubernetes und Cloud-Diensten zuständig und ermöglicht dabei die Verwaltung und Integration von Cloud-Infrastruktur durch einen Plugin-Mechanismus. [Wel24, S. 49-50]

Die **Worker Nodes** hingegen verrichten die tatsächliche Arbeitslast, auf denen auch die Applikationscontainer ausgeführt werden. Jede Worker Node beherbergt den **Kubelet-Prozess**, der als Kanal zwischen den Pods und dem API Server des Master Nodes dient, den **Kube Proxy**, der als Netzwerk-Proxy die Kommunikation zu den Pods ermöglicht und die **Container-Runtime**, die das Ausführen von Containern welche das Kubernetes Runtime Interface implementieren unterstützt. Außerdem liegt auf ihnen der Kubernetes DNS-Server für die Kommunikation innerhalb des Clusters. [Kub24c]

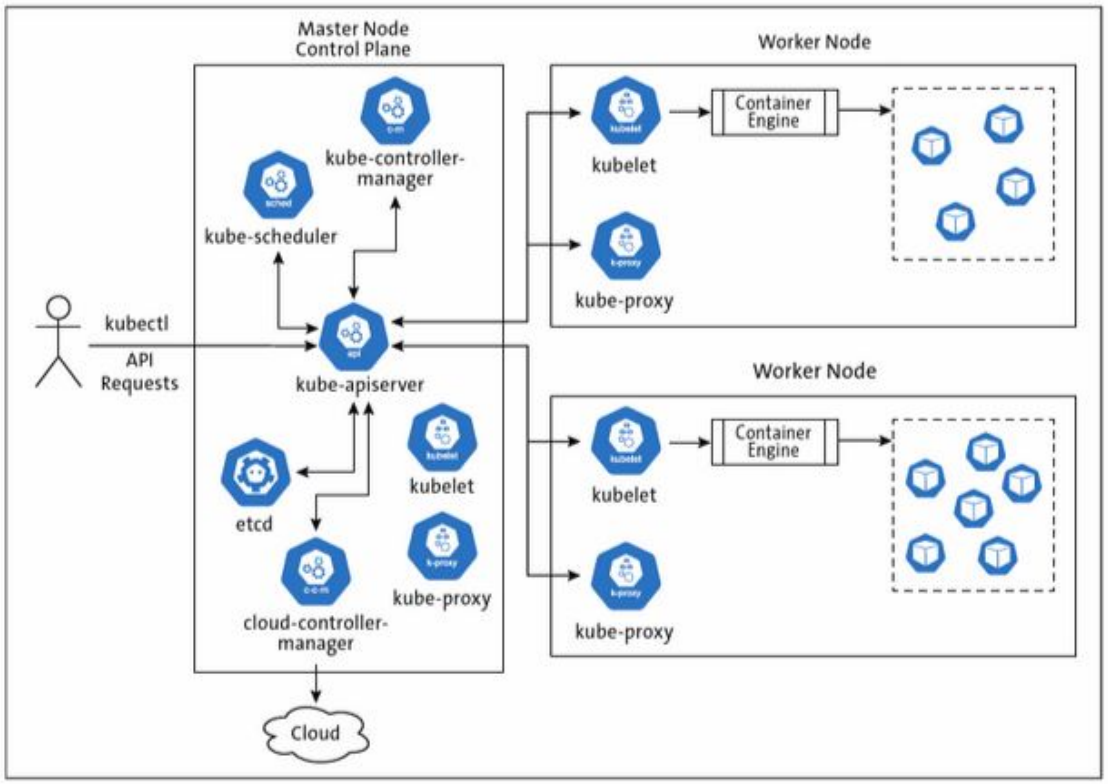

Abbildung 3.8: Kubernetes Architektur [Wel24, S. 47]

Nachdem die Architektur von Kubernetes verstanden wurde (siehe Abbildung 3.8), folgt jetzt eine Übersicht über die verschiedenen Komponenten von Kubernetes die bei Bedarf vom Entwickler konfiguriert werden müssen:

Pod

Die kleinste deploybare Einheit ist der Pod, der üblicherweise aus einem oder bei Bedarf mehreren Containern besteht, welche eine gemeinsame IP-Adresse teilen. [Ric18, Kapitel 12.4.1]

Deployment

Das Deployment definiert den Zustand der Pods und ist verantwortlich für deren Bereitstellung und Upgrades. Die eigentliche Arbeit wird dabei vom ReplicaSet erledigt, das den Soll-Zustand der Pods überwacht und bei Bedarf neue Container hoch- oder herunterfährt. [Kub24a] Beispielsweise werden hier die Anwendungs-Images und deren Anzahl konfiguriert.

Service

Ein Service stellt einer Containergruppe eine beständige IP-Adresse zur Verfügung. Dies ist wichtig, da Container jederzeit hoch und runtergefahren werden können und eine gleichbleibende Ansprech-IP für alle Container benötigt wird. Die IP-Adresse eines Kubernetes-Services kann dabei so eingestellt werden, dass sie nur innerhalb des Clusters erreichbar ist (*ClusterIP*) oder dass sie zusätzlich externen Zugriff ermöglicht, beispielsweise über *NodePort* oder einen *LoadBalancer*. [Kub24f]

In Kubernetes übernimmt der Service auch die Aufgabe der Service Discovery, indem er die Netzwerkanfragen zu den richtigen Pods weiterleitet. Die DNS-Adresse des Services kann innerhalb eines Namespaces mit seinem Namen oder innerhalb des Clusters über *[ServiceName].[Namespace].svc.cluster.local* angesprochen werden. [Wel24, S. 186]

Ingress

Das Ingress-Objekt in Kubernetes leitet eingehenden Traffic basierend auf definierten Regeln an die entsprechenden Services weiter, etwa nach URL oder Pfad. Die eigentliche Implementation von Ingress wird durch den Ingress Controller bestimmt, welcher typischerweise einen Load Balancer bereitstellt. Je nach Implementation unterstützt Ingress auch die Neuzuweisung von Pfaden und die Nutzung von TLS-Zertifikaten für HTTPS-Verbindungen. [Wel24, S. 192-193]

Volume

Volumes sind im Wesentlichen Verzeichnisse, die von einem oder mehreren Containern verwendet werden können und das Datenmanagement vereinfachen. Sie abstrahieren das tatsächliche Management der Festplatten, sodass Entwickler sich nicht mit den Details der Speicherverwaltung auseinandersetzen müssen. [Wel24, S. 284] Trotzdem ist wichtig zu verstehen, dass Kubernetes den Speicher nicht selbst bereitstellt, sondern als Abstraktion auf vorhandene Quellen wie z. B. eine Google Persistent Disk zurückgreift.

StatefulSet

Im Gegensatz zu Deployments, die Pods gleichzeitig und ohne spezifische Reihenfolge starten, skaliert und stellt das StatefulSet Pods in geordneter und voraussehbarer Weise bereit. Dies ist besonders wichtig für zustandsbehaftete Anwendungen, da jedem Pod eine eindeutige und beständige Identität zugewiesen wird, die auch bei Neustarts oder Updates erhalten bleibt. [Wel24, S. 285] Beispielsweise erhalten so Replikate eines Datenbankclusters eine konsistente Identität, um sich im Netzwerk zu finden und miteinander kommunizieren zu können. Für zustandslose Anwendungen hingegen wird das Deployment empfohlen.

ConfigMap

ConfigMap wird eingesetzt, um Konfigurationen zu speichern, die von Containern genutzt werden. Diese Konfigurationen können als Umgebungsvariablen oder Dateien in laufende Container gemappt werden. [Kub24d] Ein ähnlicher Mechanismus wird durch Secrets bereitgestellt, die sensible Daten wie Anmeldeinformationen enthalten können.

Beachtenswert ist jedoch, dass jeder mit vollständigem Lesezugriff auf den API-Server auf die Werte von erstellten Secrets zugreifen kann, was ihre Sicherheit beeinträchtigt. [Kub24e]

Fazit

Durch die automatische Replizierung von Applikationscontainern, den Failover-Mechanismen und der Möglichkeit, neue Updates ohne Ausfallzeit einzuspielen, sorgt Kubernetes für eine hohe Verfügbarkeit, kostengünstige Skalierungsmöglichkeit und Agilität. Zusätzlich sparen wir uns die Implementation einer Service-Discovery-Komponente, da diese Funktion von Kubernetes mitgeliefert wird. Aufgrund der Erfüllung dieser Anforderungsvoraussetzungen (**siehe QA/1 - QA/3**) und vorheriger Erfahrung wurde Kubernetes als Orchestrierungsplattform gewählt.

3.11.4 Bereitstellung in der Cloud

Das Bereitstellen eines Kubernetes-Clusters benötigt Ressourcen. Um die maschinelle Infrastruktur nicht selbst stellen und verwalten zu müssen wird auf Möglichkeiten des Cloud Computing zurückgegriffen. Christian Baun, Marcel Kunze, Jens Nimis und Stefan Tai definieren dabei Cloud Computing wie folgt:

„Unter Ausnutzung virtualisierter Rechen- und Speicherressourcen und moderner Web-Technologien stellt Cloud Computing skalierbare, netzwerk-zentrierte, abstrahierte IT-Infrastrukturen, Plattformen und Anwendungen als on-demand Dienste zur Verfügung. Die Abrechnung dieser Dienste erfolgt nutzungsabhängig.“ [Bau+11, S.4]

Cloud Computing bietet enorme finanzielle Vorteile, da Unternehmen keine hohen Investitionen in eigene Hardware und Software tätigen müssen und stattdessen nach einem verbrauchsabhängigen Kostenmodell zahlen. Dies ermöglicht insbesondere kleineren Unternehmen und Start-ups einen kostengünstigen Zugang zu früher exklusiven Technologien und flexibel skalierbaren IT-Ressourcen. [Rei18, S.15]

Aus diesem Grund wurde auf eine Bereitstellung in der Cloud durch die Google Cloud Platform (im Folgenden GCP) gesetzt. GCP bietet die Infrastruktur für eine hochverfügbare und skalierbare Lösung, die sich für unsere nicht-funktionalen Anforderungen eignen. Neben der guten Dokumentation und dem großzügigem Startguthaben wurden durch frühere Projektarbeiten gute Erfahrungen mit dieser Plattform gemacht.

Google Kubernetes Engine

Das Kubernetes-Cluster wird auf der Google Kubernetes Engine (im Folgenden GKE) aufgesetzt. GKE ist ein vollständig verwalteter Kubernetes-Dienst von Google Cloud, der eine schnelle Bereitstellung von Container-Anwendungen ermöglicht. Ein großer Vorteil von GKE ist die automatische Skalierung der zugrunde liegenden Infrastruktur, die die Konfiguration und Bereitstellung sämtlicher physischer Ressourcen übernimmt und diese an die Anforderungen des Systems anpasst. Die Entscheidung für GKE fiel aufgrund seiner Fähigkeit, das System schnell und unkompliziert bereitzustellen und gleichzeitig den gesamten Verwaltungsaufwand für die Infrastruktur zu minimieren. Allerdings müssen die hohen Kosten, die mit der Nutzung von GKE verbunden sind, insbesondere im Produktionsbetrieb berücksichtigt werden.

Datenbanken

Die Datenbanken für das System werden mit Cloud SQL bereitgestellt. Cloud SQL ist ein vollständig verwalteter relationaler Datenbankdienst von Google Cloud, der die Konfiguration und Verwaltung eines skalierbaren und hochverfügbaren PostgreSQL-Datenbankclusters übernimmt. Gegen eine erhöhte Gebühr werden damit die nicht-funktionalen Anforderungen an Hochverfügbarkeit und Skalierbarkeit abgedeckt. (**siehe QA/1 - QA/2**) Einzig die Redis-Datenbank wird selbst verwaltet, da die Cloud-Lösung Memorystore von Google für eine Redis-Instanz erhebliche Kosten verursacht. (Geschätzt mindestens 30 Dollar pro Monat für die niedrigste Stufe) Die Datenbank des Payment-Service wird als H2-Datenbank im Arbeitsspeicher gehalten, da diese Komponente zunächst nur ein Simulationsservice ist, der in späteren Versionen weiter ausgebaut oder ersetzt wird.

Speicher

Für die Speicherung der Bild- und Audiodateien wird der Cloud-Speicherdienst Google Cloud Storage von Google verwendet. Dabei werden Dateien als Objekte in sogenannten Buckets gespeichert, welche die grundlegenden Datencontainer des Dienstes darstellen. In den Buckets selbst herrscht eine flache Namenshierarchie, wodurch es keine klassischen Verzeichnisse gibt. Daher sollte für jedes Template ein eigener Bucket verwendet werden, weil somit jegliche komplexe Anwendungslogik für das Handhaben von Pfaden innerhalb

des Buckets wegfällt. Die Buckets können innerhalb derselben Region wie unser VPC-Netzwerk erstellt werden, was zu einer geringeren Latenz und schnelleren Upload-Zeiten unserer Daten führt.

Für den persistenten Speicher der Elasticsearch-Logs wird Google Cloud Persistent Disk verwendet. Persistent Disks sind zuverlässige Blockspeicher von Google Cloud, die Daten unabhängig von der Lebensdauer der virtuellen Server speichern. Sie lassen sich sehr einfach in Kubernetes integrieren und erfordern je nach Bedarf nur minimalen Konfigurationsaufwand.

Da nun die Grundlagen für das Deployment geschaffen wurde, folgt die Verteilungssicht des Systems.

3.11.5 Verteilungssicht

Die Verteilungssicht zeigt alle laufenden Bausteine des Systems in deren Ausführungsumgebung. Fokussiert wird also die tatsächliche physische Hardware oder virtuelle Umgebung, die für den Betrieb des Systems benötigt wird. [SH11, S.70-73] Dabei wurde das UML-Deployment-Diagramm um spezifische Kubernetes Elemente wie Ingress, Deployments, DaemonSet und StatefulSets erweitert, weil diese durch die Standard-Klassen wie Deployment Targets oder Artefakte nicht angemessen dargestellt werden können und ein insgesamt übersichtlicheres Diagramm entsteht. Außerdem wurden die Kommunikationspfade zwischen den Deployments und StatefulSets gezeichnet, da dies die Kommunikation zwischen den Artefakten besser abstrahiert.

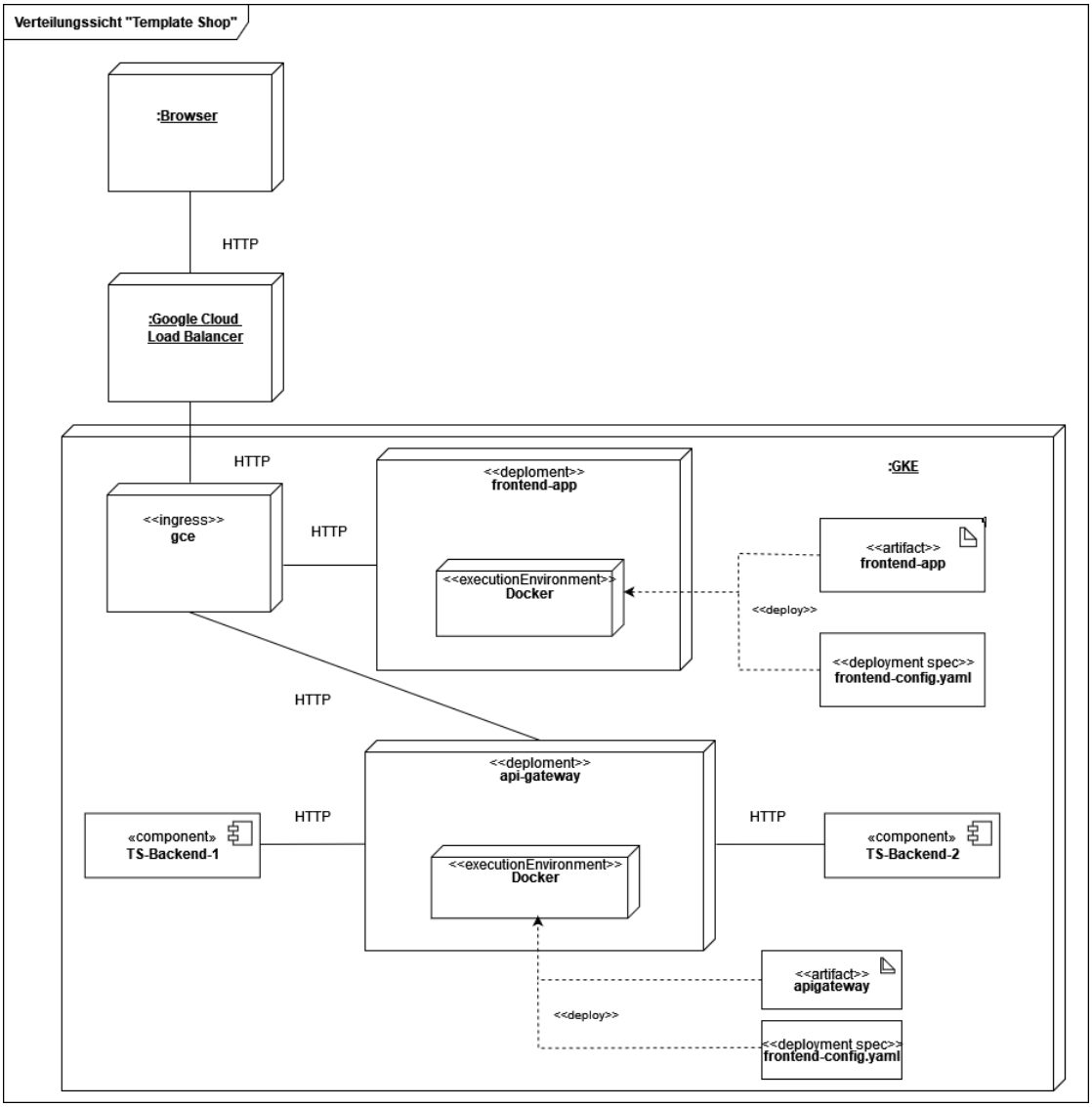

Abbildung 3.9: Verteilungssicht für den Template Shop

Abbildung 3.9 zeigt das erweiterte UML-Deployment-Diagramm für unsere Templateshop-Anwendung. In dieser Architektur stellt der Kubernetes Ingress-Controller einen Load Balancer bereit, der es ermöglicht, dass Clients über ihren Browser auf die Frontend-Anwendung zugreifen. Alle Anfragen vom Browser werden über den Ingress-Controller an das API-Gateway weitergeleitet, welches die Anfragen anschließend an die entsprechenden Backend-Services verteilt. Die auszuführenden Artefakte sind als Docker-Images in der Artifact-Registry der Google Cloud Platform gespeichert. Zur besseren Übersichtlichkeit wurden Details wie die Artifact-Registry, Pods, zusätzliche Konfigurationen wie

ConfigMaps für Datenbankinformationen und Secrets in diesem Diagramm nicht dargestellt. Die Konfiguration des Frontends legt die externe URL für den Ingress-Controller fest, während die API-Gateway-Konfiguration Cross-Origin Resource Sharing (CORS) für den Ingress-Controller aktiviert. Die beiden TS-Backend-Komponenten werden in separaten UML-Deployment-Diagrammen in Abbildung 3.10 und 3.11 detaillierter dargestellt.

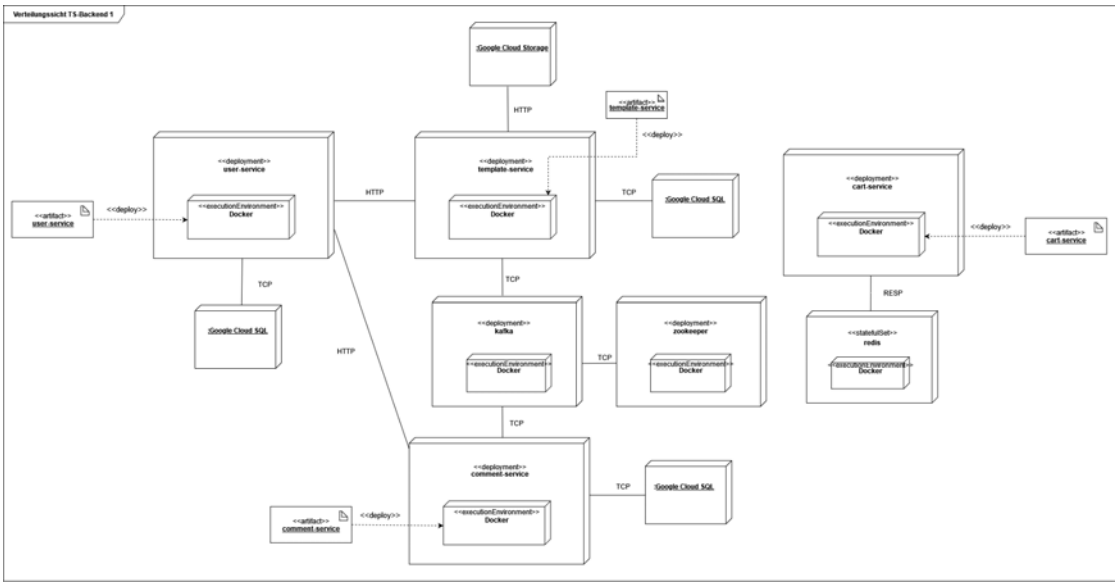

Abbildung 3.10: Verteilungssicht für das Template Shop-Backend 1

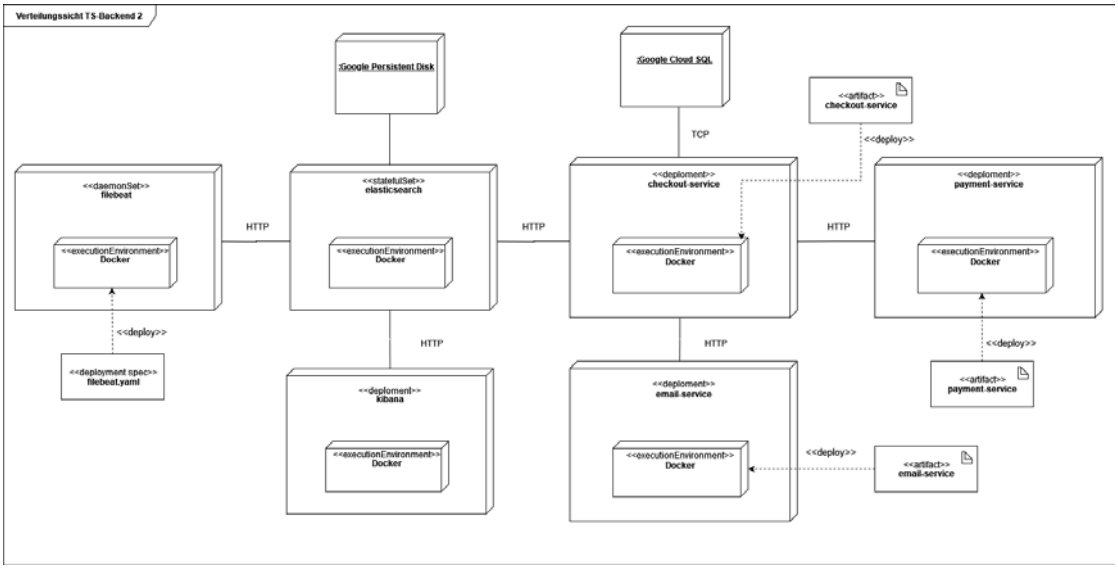

Abbildung 3.11: Verteilungssicht für das Template Shop-Backend 2

In Abbildung 3.11 wird Filebeat als *DaemonSet* bereitgestellt. Ein Kubernetes *DaemonSet* stellt sicher, dass auf jeder Node im Kubernetes-Cluster ein Pod mit einer Instanz von Filebeat ausgeführt wird. Dadurch können die Docker-Logs von allen Nodes gesammelt und an Elasticsearch weitergeleitet werden. Zur besseren Übersicht wurde in der Abbildung nur ein *DaemonSet* dargestellt, das stellvertretend für alle Filebeat-Instanzen auf den Nodes steht.

4 Realisierung

4.1 Backend-Framework

Das Backend bietet Services und Funktionen an, um Anfragen von Clients zu bearbeiten. Hierzu gehört die Verarbeitung von Geschäftsfunktionen als auch die Bereitstellung von Daten über APIs. Aufgrund seiner Vertrautheit und der Möglichkeit, sämtliche Technologien, die in der Bausteinsicht definiert wurden, leicht ins System zu integrieren, wurde Spring Boot zur Implementation des Backends verwendet.

Spring ist ein Java-Framework, das zur Entwicklung von Unternehmensanwendungen genutzt wird und sich in vier Hauptmodule gliedert. Diese Module decken verschiedene Funktionalitäten ab: von der Verwaltung und Konfiguration von Beans über den Zugriff auf Datenbanken bis hin zu Web-Schnittstellen und Performance-Überwachung. [Gol20, S.11-12] Spring Boot selbst ist dabei eine vorkonfigurierte Spring-Anwendung, die sofort gestartet werden kann und die Entwicklung mit dem Framework vereinfacht. [Gol20, S.29]

Zwei wichtige Komponenten in der Spring Architektur sind der Spring Container und seine Beans: Der Spring Container sorgt für die korrekte Initialisierung und Ablauf der Anwendung. Außerdem steuert er den Lebenszyklus und die Verknüpfung jeglicher Beans innerhalb des Containers. Beans sind dabei im Grunde nur Java-Objekte, die per Annotationen oder XML-Deklaration markiert werden, um so vom Container erkannt und verwaltet zu werden. [Gol20, S.22]

Der Vorteil von Beans gegenüber Java Objekten besteht in den zusätzlichen Funktionen, die der Spring Container ihnen bietet: Durch Dependency Injection werden automatisch Abhängigkeiten zwischen Beans aufgelöst, ohne dass eine manuelle Initialisierung nötig ist. Dieses Konzept der Inversion of Control, nämlich dass ein Objekt seine Abhängigkeiten nicht vom Entwickler, sondern von jemand anderem erhält (z. B. den Spring Con-

tainer), sorgt für eine lose Kopplung zwischen den Komponenten, da beispielsweise auch beliebige Implementationen von Interfaces injiziert werden können. [Gol20, S.64-65]

Standardmäßig existiert zur Laufzeit nur eine Bean, die vom Spring-Container verwendet wird. Ist diese Bean zustandslos, kann die Performance des Systems durch Skalierung und Caching verbessert werden, da mehrere Threads gleichzeitig auf das Objekt zugreifen können, ohne sich gegenseitig zu behindern. Dies ist aber nur bei ausreichender Infrastruktur gegeben. Möchte man zur Laufzeit mehrere Beans erstellen, kann dies durch unterschiedliche *Scopes* in Spring Boot konfiguriert werden. [Gol20, S.82-86]

4.1.1 Services

Fast jeder Service wird durch einen Controller-Bean, einen Service-Bean und einen Repository-Bean definiert. Im Controller wird mittels Spring MVC ein REST-Endpunkt erstellt, der über das HTTP-Protokoll die CRUD-Funktionalitäten der Entitäten unserer Domäne anbietet. Dabei erleichtert Spring MVC durch Annotationen die Erstellung eines Servlet-Containers, welcher die HTTP-Anfragen eines Fremdsystems entgegennimmt und beantwortet. [Gol20, S.12]

Ein Service enthält dabei die Logik für CRUD-Operationen und nutzt im Repository-Bean Spring Data sowie Hibernate, um mit der Datenbank zu kommunizieren.

Spring Data vereinfacht die Implementierung der Datenzugriffsschicht, indem es eine abstrahierte Schnittstelle zur Verfügung stellt, die die Integration verschiedener Datenbanktechnologien ermöglicht. Hibernate hingegen, als weitverbreitete JPA-Implementierung, sorgt für das objektrelationale Mapping, indem es Java-Objekte mit Datenbanktabellen verbindet. [Gol20, S.94-95]

Diese Kombination stellt die Kommunikation zwischen der Anwendungs- und Persistenzschicht her und unterstützt die Nutzung verschiedener Datenbanklösungen, ohne dass der übrige Code angepasst werden muss. Dies trägt zur Wartbarkeit des Systems bei, welche wir als Qualitätsanforderung definiert haben. (**siehe QA/4**)

Im Folgenden werden einige Besonderheiten einzelner Services aufgezählt:

Cart-Service

Wie im vorherigen Kapitel erläutert, wurde für die Warenkorb-Implementierung ein Controller unter Verwendung von Redis und Spring Session entwickelt. Alle Warenkorbinformationen werden als String in Redis gespeichert. Der Schlüssel für diesen String wird durch die Konkatenation der im Cookie gespeicherten Session-ID und einer vordefinierten Zeichenkette erstellt. In der aktuellen Version des Prototyps wird jedem Client-Browser eine Session-ID zugewiesen, die so die Rolle eines Gastes repräsentiert. Während des Anmelde- oder Abmeldevorgangs bleiben die Warenkorbinformationen somit unverändert.

Template-Service

Neben der CRUD-Funktionalität für Templates und Tracks bietet der Template-Service einen Upload-Endpunkt an, welcher Multipart-Anfragen annimmt um Dateien wie Bilder und Audiospuren über die Google Cloud Storage API hochzuladen. Über einen Download-Endpunkt können die hochgeladenen Tracks heruntergeladen und vom Frontend anschließend für die visuelle Darstellung verwendet werden.

Checkout-Service

Neben der Verwaltung von Bestellungen und der Kommunikation mit dem Payment- und E-Mail-Service nutzt der Checkout-Service einen Elasticsearch-Client, um Bestellungen über eine REST-API in Elasticsearch zu indexieren. Diese Bestellungen können dann in einem Dashboard mit Kibana visualisiert werden. Alternativ bietet der Service einen Endpunkt zum Abfragen der Bestellungen an, damit das Frontend diese dem Admin auf anderem Wege zur Verfügung stellen kann.

4.2 Sicherheit

Alle Autorisierungs- und Authentifizierungsmechanismen sowie Sicherheitseinstellungen werden mit Spring Security konfiguriert und implementiert, da dies das etablierte Sicherheitsframework für Spring Anwendungen ist.

Jeder Zugriff auf geschützte Ressourcen unterliegt der Kontrolle eines selbst implementierten Filters. Ein Filter in Spring Security ist eine Komponente, die in den Anfragezyklus eingreift und benutzerdefinierte Logik zur Verarbeitung von Anfragen oder Antworten bereitstellt. Dieser Filter validiert das mitgesendete JSON Web Token (JWT) und setzt den *Principal* bzw. die *Authentication* in den *Security-Context*, wobei die Informationen für den *Principal* aus dem JWT übernommen werden. Der *Principal* repräsentiert dabei den authentifizierten Benutzer und ist Teil der *Authentication*, welche zusätzliche Informationen über dessen Rollen und Berechtigungen trägt. Der *Security-Context* beinhaltet Informationen über den Authentifizierungsstatus einer Anfrage und enthält Zugriff auf die *Authentication*.

Durch den *Security-Context-Holder*, der den *Security-Context* für den aktuellen Thread beinhaltet, kann nun der Authentifizierungs- und Autorisierungsstatus der aktuellen Anfrage ermittelt werden. [Gol20, S.161-162] Mittels Annotationen werden die Endpunkte dann mit bestimmten Rollen versehen, um den Zugriff entsprechend einzuschränken. Zum Beispiel ermöglicht die Zuweisung der Admin-Rolle, dass nur Benutzer mit dieser Rolle die Berechtigung haben, Templates anzulegen oder zu löschen. Annotationen ermöglichen auch die Festlegung von Regeln, die sicherstellen, dass ein Benutzer nur seinen eigenen Account löschen kann und nicht von anderen.

Die Validierung des JWT erfolgt mithilfe der Java-JWT Library, welche die Implementierung aller JWT-Funktionalitäten bereitstellt. Um den JWT zu validieren, muss der Service über eine REST-API den Public Key des User-Service anfragen.

Alle weiteren Sicherheitseinstellungen, die z. B. CORS und CRSF Sicherheit betreffen, werden programmatisch konfiguriert.

4.3 Tests

In der Softwareentwicklung ist Testen der Vorgang, der sicherstellt, dass Anwendungen wie erwartet funktionieren. [Ric18, Kapitel 9] Dabei gibt es verschiedene Testarten, die unterschiedliche Aspekte einer Anwendung prüfen und größtenteils automatisiert ausgeführt werden können:

Unit-Tests prüfen die Korrektheit von kleinen Teilen eines Services, wie z. B. einzelnen Klassen. Integrationstests testen die Interaktion von Services mit Infrastrukturkomponenten wie Datenbanken und anderen Applikationsdiensten. Komponententests prüfen

hingegen einen ganzen Service, während End-to-End-Tests die gesamte Applikation kontrollieren. [Ric18, Kapitel 9.1.1]

Aus zeitlichen Gründen wurde auf umfangreiche Integrationstests mit Datenbanken und Applikationsdiensten verzichtet, weil die korrekte Funktionalität der Infrastrukturkomponenten in den Komponententests und End-to-End-Tests ersichtlich wird.

4.3.1 Konfiguration

Um Produktionsdaten nicht zu beeinträchtigen, können für die Tests verschiedene Spring-Profiles verwendet werden. Spring-Profiles bieten die Möglichkeit, spezifische Konfigurationen für unterschiedliche Umgebungen bereitzustellen. Damit kann flexibel zwischen Applikationsdiensten und Interface-Implementationen gewechselt werden, um eine Komponente in verschiedenen Umgebungen zu testen.

4.3.2 Komponententests

Für die isolierten Komponententests des Backends wird JUnit und Mockito verwendet. Mockito ermöglicht das Erstellen von Mock-Objekten, welche externe Abhängigkeiten simulieren und das Verhalten von Methoden überwachen können. Dies bedeutet, dass wir ohne eine Implementation einer Abhängigkeit wie z. B. eines anderen Services, deren Funktion wie bestimmte Rückgabewerte einer Methode bestimmen können.[Ric18, Kapitel 9.2.4] Da unsere Services des Backends so geschnitten sind, dass sie kaum Abhängikeiten zu anderen Services haben, dient Mockito eher dazu den asymmetrischen Schlüssel für das Backend zu schreiben oder Exceptions zu simulieren. Beispielsweise kann man mit der SpyBean-Annotation eine Komponente von Spring Boot in seiner normalen Implementation im Spring Container benutzen, aber einzelne Methoden ändern.

Mit MockMvc lassen sich Controller-Anfragen und -Antworten testen, ohne dass die gesamte Spring Boot-Umgebung oder echte Netzwerkverbindungen erforderlich sind. [Ric18, Kapitel 9.2.5] Mit dem in Spring Boot integrierten Jackson-Mapper können wir hierbei Java-Objekte als JSON versenden und mithilfe der JSONPath Bibliothek die erwarteten Ergebnisse prüfen.

4.3.3 Komponententests in Integration

Für die Durchführung von Tests mit mehreren Services oder Diensten kommt die Testcontainer-Bibliothek zum Einsatz. Testcontainer ermöglicht das Initiieren und Beenden von Docker-Containern direkt innerhalb der Testfälle. Beispielsweise kann so schnell ein Redis-Container für die Unit-Tests und Integrationstests des Cart-Service hochgefahren werden. Mittels der Bibliothek wurde dann die Kommunikation zwischen E-Mail-, Checkout- und Payment-Service, Template- und Comment-Service und die jeweiligen Abhängigkeiten zum User-Service getestet. Dabei wurde in komplexeren Netzwerken wie beispielsweise Kafka mit docker-compose gearbeitet, damit die Container während des Tests problemlos miteinander kommunizieren konnten. Wichtig ist hier, die zugeordneten Ports von Testcontainer abzufragen, da die Ports der Docker-Container dynamisch zugewiesen werden.

4.3.4 End-To-End Tests

Die End-To-End-Tests des Systems wurden für jeden Use Case manuell mit docker-compose, später lokal in Kubernetes mit Minikube und schließlich auf der Google Cloud Platform in der Produktionsumgebung ausgeführt. Aufgrund zeitlicher Gründe und des Aufwands, Frontend mit Selenium oder Cypress zu automatisieren, wurde sich für den manuellen Test-Ansatz entschieden.

4.4 Frontend-Framework

Das Frontend bildet die Weboberfläche einer Anwendung, über die der Benutzer mit den Funktionen des Backends interagieren kann. Die Wahl hierfür fiel auf das Javascript-Framework Vue.Js, welches als modernes, komponentenbasiertes Framework eine flexible und einfache Frontend-Entwicklung bietet.

Vue.js wurde von Evan You entwickelt, der zuvor bei Google mit AngularJS an verschiedenen Projekten gearbeitet hatte. Ziel war es, die besten Aspekte von AngularJS zu extrahieren und in ein leichteres Framework zu integrieren. Seit den ersten Commits 2013 wird Vue.js von einer ständig wachsenden Community entwickelt. [Dei22, S.2]

In Vue.js (im Folgenden Vue) wird die Weboberfläche in unabhängige, wiederverwendbare Komponenten unterteilt, wobei jede Komponente ihren eigenen Zustand hält. Durch die Verwendung von Props und Events können dabei Daten von übergeordneten zu untergeordneten Komponenten übertragen und verändert werden. Da sich die Verschachtelung der Komponenten in der Anwendung in Grenzen hielt, wurde auf ein Zustandsmanagementsystem verzichtet.

Zustandsmanagement beschreibt dabei die Verwaltung und zentrale Speicherung von Zuständen einer Anwendung, um diese über Komponenten hinweg zu teilen. [Dei22, S.165-166] Wer aber mit komplexeren Zustandsabhängigkeiten über Komponenten hinweg hantiert, sollte zu Lösungen wie Vuex oder Pinia zugreifen.

Aufbauend auf MVC verwendet Vue das MVVM-Muster. Dabei fungiert das ViewModel als Bindeglied zwischen der View und Model, indem es Informationen austauscht und Methoden des Models aufruft. Dies ermöglicht eine deklarative Datenbindung, bei der keine separaten Controller-Instanzen erforderlich sind, wodurch der manuelle Zugriff auf das DOM stark reduziert wird. Zusätzlich stellt das ViewModel der View öffentliche Eigenschaften und Methoden zur Verfügung, die an Steuerelemente gebunden werden, um Inhalte auszugeben und UI-Ereignisse weiterzuleiten. [Ste19, S.43]

Außerdem verwendet Vue einen virtuellen DOM, bei dem eventuelle Änderungen nicht direkt am DOM vorgenommen werden, sondern zunächst in einer Kopie des DOM als interne JavaScript-Datenstruktur vorliegen. Diese Änderungen werden dann mit dem ursprünglichen DOM verglichen und erst danach zusammengefasst auf den realen DOM übertragen. [Ste19, S.10-11]

Durch diese Eigenschaften eignet sich Vue sehr gut für die Entwicklung von reaktiven Single Page Applications, also solchen, die aus einem einzigen HTML-Dokument bestehen und deren Inhalte dynamisch durch JavaScript nachgeladen werden. [Ste19, S.70]

Der Hauptgrund für die Wahl von Vue war ähnlich wie bei Spring Boot die existierende Vorerfahrung mit dem Framework. Aufgrund der Ähnlichkeiten beliebter Frontend-Frameworks wäre auch jede andere Wahl ausreichend gewesen, um die Anforderungen für das System zu implementieren.

4.4.1 Datenpersistenz

Die meistverwendeten Möglichkeiten, Informationen im Browser zu persistieren, sind Cookies und der *localStorage* bzw. *sessionStorage*.

Cookies sind kleine Textdateien, die vom Server an den Browser gesendet und auf der Festplatte des Benutzers gespeichert werden. Sie enthalten oft Informationen wie Benutzereinstellungen oder Sitzungsdaten und haben eine begrenzte Größe sowie festgelegte Gültigkeitsdauer.

LocalStorage und *sessionStorage* sind Teile des Web Storage API und bieten uns eine alternative Möglichkeit, Daten im Browser zu speichern. *LocalStorage* speichert Daten dauerhaft, während Daten vom *sessionStorage* nur für die Dauer einer Browsersitzung gültig sind. Im Gegensatz zu Cookies werden Informationen aus *localStorage* und *sessionStorage* nicht bei jeder HTTP-Anfrage an den Server geschickt.

Für die Speicherung des JWTs im Frontend wurde das *localStorage* verwendet, da es damit sehr einfach ist die benötigte Funktionalität für die Anwendung zu implementieren. Weil es keine besonderen Sicherheitsanforderungen bezüglich des Frontends gibt, können Sicherheitsbedenken wie potenzielle Anfälligkeiten für Cross-Site Scripting mit dem Arbeiten des *localStorage* vorerst außer Acht gelassen werden. Wichtig wäre zu wissen, dass Informationen im *localStorage* unverschlüsselt gespeichert werden, weshalb sensible Daten verschlüsselt werden sollten.

Spring Session speichert seine Session-IDs standardmäßig in Cookies, weshalb auch diese im System verwendet werden.

4.4.2 Routing

Für das Routing, also der Verknüpfung von URLs zu bestimmten Ansichten, wird Vue Router verwendet. Vue Router ist der offizielle Router von Vue-Anwendungen und arbeitet mit deklarativen Routenkonfigurationen, die sich einfach mit Vue-Komponenten verbinden lassen. Wir definieren dabei welche Komponenten auf einer bestimmten URL anzeigt werden sollen und können diese dynamisch für jedes Template konfigurieren. Im Ergebnis erhalten wir eine Seite mit einer einzigen *router-view*, welche reaktiv alle Komponenten einer bestimmten URL rendert, ohne dass ein Neuladen der Seite benötigt wird.

Außerdem kann mit *NavigationGuards* überprüft werden, ob der Benutzer autorisiert ist, eine URL anzusteuern.

4.4.3 Design

Für das Design wurde das weitverbreitete CSS-Framework Bootstrap verwendet, welches ursprünglich von Twitter entwickelt wurde. Es bietet vordefinierte Klassen und Komponenten wie *buttons* und Formulare, um schnell eine Benutzeroberfläche aufsetzen zu können. Darüber hinaus bietet Bootstrap JavaScript-Komponenten wie *modals* und *dropdowns* an, welche die Implementierung von dynamischen Funktionen beschleunigen.

4.4.4 Audiospur

Für die Darstellung der Audiospuren wurde die JavaScript-Bibliothek Wavesurfer.js verwendet. Diese Bibliothek ermöglicht sowohl die Wiedergabe als auch die Visualisierung von Audiodateien, indem sie diese in anpassbaren Wellenformen darstellt. Mithilfe der bereitgestellten Methoden können wir verschiedene Informationen der Audiospur abfragen und Event-Handler für spezifische Ereignisse konfigurieren. Beispielsweise kann so durch das Tracking der Zeit einer Audiospur im Frontend immer der korrekte Zeitpunkt für das Erstellen eines Kommentars angezeigt werden.

4.4.5 Kommentar-Icons

Mit *tooltips* von Bootstrap werden Kommentare dynamisch innerhalb einer Kommentarbox angezeigt, sobald der Benutzer mit der Maus über die Icons eines Kommentars der Audiospur fährt. Die Icons werden dabei aus den ersten beiden Buchstaben des Benutzernamens zusammengestellt. Die Hintergrundfarbe der Icons wird auf Basis des Alphabets gleichmäßig auf sechs Farben verteilt und die Farbe entsprechend dem Anfangsbuchstaben des Benutzernamens gewählt.

Für die richtige Positionierung der Icons auf der Audiospur wird die aktuelle Zeit der Audiospur durch die Gesamtzeit geteilt und mit der Breite des Containers multipliziert. Dadurch entsteht ein erster Wert auf der x-Achse, der proportional zur Dauer der Audiospur und zur Containergröße ist. Anschließend wird dieser Wert um den Offset des

Containers ergänzt, um den Platz links vom Container zu berücksichtigen. Für die vertikale Position der Icons wird der vertikale Offset des Containers mit einem konstanten Wert addiert und anschließend mit dem Scroll-Offset des Bildschirms berechnet. Der Scroll-Offset ist wichtig, um sicherzustellen, dass die Icons korrekt positioniert werden, falls der Benutzer die Webseite vor dem Laden der Icons scrollen sollte. Weil in der ersten Version des Prototyps mit absoluten Werten gerechnet wird, verhält sich diese Lösung nicht responsiv.

4.4.6 Kommentare

Zum Erstellen und Anzeigen von Kommentaren wurden *modals* von Bootstrap verwendet. *Modals* sind Dialogfenster, die über dem Hauptinhalt eingeblendet werden, ohne die aktuelle Ansicht zu verlassen. Dies verbessert die Übersichtlichkeit der Anwendung, da die Kommentarfunktionalitäten vom Benutzer flexibel geöffnet und geschlossen werden können, ohne dass andere Elemente wie die Audiospuren verschoben werden. Die Kommentare werden dabei jeweils durch die Track-ID im Backend gefiltert, um nur die Kommentare einer zugehörigen Audiospur anzuzeigen. Die Icons werden auch hier eingesetzt, indem sie neben den Benutzerkommentaren angezeigt werden.

4.4.7 Upload

Für den Upload wird ein Formular bereitgestellt, das die Metadaten sowie die Bild- und Audiodateien des Templates enthält. Mithilfe der Datenbindung in Vue werden die Eingaben durch reguläre Ausdrücke überprüft, um sicherzustellen, dass in den Metadaten ausschließlich alphanumerische Zeichen verwendet werden. Beim Absenden des Formulars wird jede Datei dann einzeln über eine Upload-Komponente an den Upload-Controller des Backends übertragen. Während es möglich wäre, den gesamten Upload als Paket in einer einzigen Anfrage zu senden, um so die Netzwerklast zu reduzieren, ist der Vorteil der Einzeldatei-Übertragung eine Verfolgung des Upload-Fortschritts jeder individuellen Datei. Dies verbessert die Fehlerbehandlung, da Probleme mit einzelnen Dateien sofort erkannt werden können.

4.4.8 Weitere Frontend-Komponenten

Alle weiteren Frontend-Komponenten besitzen keine komplexere Logik, sondern dienen entweder der Darstellung sämtlicher Informationen, die vom Backend geliefert werden, oder der Aufnahme, Überprüfung und Übermittlung von Daten über Formulare an die Services. Mit JSON Web Tokens (JWT), die im *localStorage* gespeichert sind und der Axios-Bibliothek können HTTP-Anfragen an die entsprechenden REST-APIs gesendet werden, um die Funktionalitäten des Frontends zu implementieren.

4.5 Betrieb

Für einen einfachen Betrieb wurde ein Cluster auf GKE im Autopilot-Modus erstellt. Neben der automatischen Skalierung und Konfiguration der Serverknoten übernimmt GKE in diesem Modus die vollständige Verwaltung der Cluster-Infrastruktur. Zudem wurde für eine verbesserte Sicherheit das Cluster in einem privaten VPC-Netzwerk in Frankfurt bereitgestellt. Der Standort der Server hat dabei in diesem Prototypen keine Relevanz, könnte jedoch in zukünftigen Versionen für gesetzliche Vorgaben wie der DSGVO wichtig werden.

Weil es keine konkreten Vorgaben für die Anzahl paralleler Anfragen an die Cloud SQL-Instanzen gibt, wurden diese auf die niedrigsten Ressourcen-Einstellungen konfiguriert, um Kosten zu sparen.

Um die Hochverfügbarkeit zu verbessern, können die Services, die mit Cloud SQL verbunden sind, zusätzlich durch mehrere Replikate bereitgestellt werden. Eine automatische dynamische Skalierung basierend auf CPU- und Ressourcennutzung wurde noch nicht eingerichtet, weshalb der Prototyp zurzeit nur manuell skaliert werden kann.

Der Zugriff von Containern innerhalb der GKE-Umgebung auf Google Cloud Services erfolgt über *Workload Identities*. Dabei wird jedem Kubernetes-Pod die Identität eines Google Service Accounts zugewiesen, der über individuell festgelegte Rollen und Berechtigungen verfügt. Auf diese Weise können die Pods ohne die Notwendigkeit von expliziten Schlüsseln auf Google Cloud Services zugreifen. Dies vereinfacht die Authentifizierung und erhöht die Sicherheit, indem die Verwaltung und Konfiguration von langfristigen Anmeldeinformationen entfällt.

4.6 Hürden der Realisierung

Während der Realisierung traten einige erwähnenswerte Probleme auf, deren Lösungen für andere Entwickler in ähnlichen Situationen hilfreich sein könnten:

4.6.1 Spring Session-Tests

Obwohl Spring Session bei den lokalen Tests Cookies an das Frontend sendete, konnte das Frontend diese nicht speichern. Der Grund dafür war, dass der CORS-Filter genauer eingestellt werden musste. Das Zulassen aller CORS-Anfragen mit einem Wildcard-Operator funktioniert nicht, da für die Speicherung von Cookies der tatsächliche Host genau spezifiziert werden muss.

4.6.2 Elasticsearch

Bei den manuellen Tests mit Elasticsearch kam es aufgrund unzureichenden Speicherplatzes auf der Festplatte zu Problemen, weil Elasticsearch ab einem bestimmten Grenzwert automatisch alle Schreiboperationen blockiert hatte. Um das Problem zu beheben, musste zunächst ausreichend Speicherplatz auf der Festplatte freigemacht werden und anschließend die Schreibrechte in Elasticsearch entweder direkt im Terminal des Docker-Containers oder über die ENTRYPOINT-Anweisungen in der Dockerfile wiederhergestellt werden.

4.6.3 Minikube

Um Minikube (v1.23.2) lokal testen zu können, muss der Docker-Daemon von Minikube zunächst auf den Docker-Daemon der lokalen Maschine zeigen. Auf einem Windows-System lautet der Befehl dazu:

```
@FOR /f "tokens=*"%i IN ('minikube -p minikube docker-env') DO @%i
```

Anschließend können die lokalen Images in Minikube geladen werden. Dabei ist es wichtig, dass die Kubernetes-Deployment-Dateien mit der *IfNotPresent* Pull-Policy ausgestattet werden. Andernfalls würde Minikube versuchen, die Images aus dem Docker Hub zu ziehen.

Endpunktvariablen sollten nicht mit *[name].endpoint* definiert werden, da der Begriff *endpoint* bereits von Minikube verwendet wird. Dies kann sonst zu Konflikten und Fehlern bei der Service-Discovery führen.

4.6.4 Filebeat auf GKE

Während Filebeat lokal und in Minikube über den Pfad */var/lib/docker/containers* auf Docker-Logs zugreifen konnte, ist dieser Zugriff im GKE-Autopilot-Modus gesperrt. Daher wurde zur Log-Sammlung statt Filebeat, eine Kombination aus Logstash und Cloud Logging benutzt. Cloud Logging ist ein von Google verwalteter Dienst welcher alle Logs aus der GKE-Umgebung sammelt und diese in der Google Cloud Console zur Verfügung stellt. Mithilfe eines definierten Log-Routers werden sämtliche Logs auf die relevanten Services gefiltert und in einem Pub/Sub Topic hochgeladen. Anschließend wird ein Logstash-Container bereitgestellt, welcher diesen Topic abonniert und sämtliche Logs an Elasticsearch weiterleitet. Logstash ist ähnlich wie Filebeat eine mit Elasticsearch einfach integrierbare Datenverarbeitungspipeline und wurde aufgrund seiner Plugins gewählt, die das Abonnieren von Pub/Sub-Topics erleichtern. Auch wenn das native Cloud Logging allein eine einfache Beobachtbarkeit ermöglicht hätte, lag die Motivation hinter dieser Anpassung darin, die bestehende Elasticsearch-Architektur beizubehalten.

4.6.5 Verbindung von GKE zu Public Cloud SQL

Auch wenn eine Cloud SQL-Instanz mit öffentlicher IP-Adresse einfach lokal über Credentials oder Cloud SQL Auth-Proxy erreichbar ist, ist diese Verbindung mit einem privaten GKE-Cluster nicht ohne Konfigurationen möglich. Um die Kommunikation hier zu vereinfachen, sollte der Instanz eine private IP-Adresse hinzugefügt werden, welche sich im gleichen VPC-Netzwerk befindet. Anstatt für die Verbindung nun mit einer Client-Bibliothek für Cloud SQL zu arbeiten, kann hier nun wie lokal üblich die private IP-Adresse und der Port der Instanz benutzt werden.

4.6.6 Verbindung von Ingress zum API-Gateway

Die aktuelle Ingress-Implementierung GCE hatte Schwierigkeiten mit dem API-Gateway zu kommunizieren, was zu der Fehlermeldung *502 Bad Gateway* führte. Laut den Ingress-Logs wurde das API-Gateway nicht als gesunder Service erkannt. Die Lösung bestand

darin, mithilfe von Spring Boot Actuator einen Health-Check-Endpunkt zu erstellen und den Google Cloud Load Balancer über die Health-Check-Einstellungen in GCP auf diesen Endpunkt zu konfigurieren. Dadurch konnte Ingress das API-Gateway als gesunden Service erkennen und die Anfragen korrekt weiterleiten.

4.6.7 Interaktion der Session-Cookies mit dem Browser

In der lokalen Entwicklungsumgebung traten keine Probleme mit den Session-Cookies von Spring Session auf, während in der Produktionsumgebung Session-Cookies vom Browser nicht gespeichert werden konnten. Das Problem war, dass der Browser Cookies nur über HTTPS-Verbindungen akzeptiert, wenn sie über Cross-Origin-Anfragen gesendet werden. In der lokalen Minikube-Umgebung gab es wahrscheinlich keine Schwierigkeiten, weil Minikube in der Regel nur mit *localhost* arbeitet und keine Cross-Origin-Anfragen über das Internet gemacht werden. Um das Problem zu beheben, musste Ingress mit einer eigenen Domäne konfiguriert und ein Zertifikat von Google für diese Domäne ausgestellt werden. Dadurch konnte zwischen Ingress und den verbundenen Services eine gesicherte HTTPS-Verbindung aufgebaut werden.

Danach gab es jedoch zusätzliche Probleme: Obwohl Vue einen positiven HTTP-Statuscode zurückgab, wurde ein *Invalid Host Header* angezeigt. Dies lag daran, dass der *webpack-dev-server* von Vue standardmäßig nur auf localhost eingestellt ist. Daher mussten zuerst die entsprechenden Domains für das Frontend freigeschaltet werden. Danach gab es wieder einen *502 Bad Gateway* und einen ungesunden Zustand des Frontend-Services. Nach einigen Anpassungen und Experimenten mit verschiedenen Health-Check-Einstellungen löste sich das Problem nach langer Zeit schließlich von selbst, als auf Standardeinstellungen zurückgesetzt wurde. Es ist schwierig zu sagen, was genau den Fehler behoben hat, da es kein klares Feedback darüber gab, welche Änderungen etwas bewirkten und welche nicht.

5 Evaluation

Nach der Realisierung des Systems wird untersucht, inwiefern dieses die funktionalen und nicht-funktionalen Anforderungen erfüllt und ob man bestimmte Aspekte hätte besser umsetzen können.

5.1 Funktionale und nicht-funktionale Anforderungen

Die Unit- und Komponententests sowie die manuellen End-to-End-Tests der Use Cases haben gezeigt, dass alle funktionalen Anforderungen des Prototyps erfüllt wurden. Dennoch könnten zusätzliche Tests notwendig sein, um noch unentdeckte Szenarien abzudecken und die Funktionalitäten des Systems beispielsweise in Grenzbereichen zu gewährleisten.

Bezüglich der nicht-funktionalen Anforderungen hat die Wahl von Kubernetes zu einem leicht skalierbaren System geführt, das bisher jedoch nur manuell skaliert werden kann. Für eine automatische dynamische Skalierung müssen wir Autoscaler von Kubernetes implementieren, die die Anzahl der Replikate eines Deployments basierend auf der aktuellen CPU- und Speicherauslastung automatisch anpassen.

Obwohl Lastenverteilung und *rolling updates* theoretisch eine hohe Verfügbarkeit gewährleisten, ist das System derzeit nicht resilient gegenüber Netzwerkfehlern, was die Verfügbarkeit beeinträchtigt. Um dies zu verbessern, sollten in Kubernetes *liveness* und *readiness probes* implementiert werden, um fehlerhafte Services zu identifizieren. Zusätzlich sollten Services wie das API-Gateway mit Circuit Breakers ausgestattet werden, um schlecht reagierende Services zu ignorieren. Auch sind der ELK-Stack und der Redis-Container derzeit die Engpässe des Systems, da für sie noch keine Cluster konfiguriert wurden, was bei extrem vielen Anfragen zu Problemen führen kann. Zudem haben wir noch keine Lasttests durchgeführt, weshalb wir derzeit nicht wissen, wie das System unter

Volllast reagiert. Ein weiteres Problem besteht darin, dass die Schlüsselpaare des User-Service zurzeit im Arbeitsspeicher gespeichert werden. Bei mehreren Replikaten könnte ein Service Schwierigkeiten haben einen JWT zu verifizieren, wenn eine Anfrage von einem Replikat bearbeitet wird, das einen anderen öffentlichen Schlüssel verwendet. Zudem kann ein Service seinen JWT nicht mehr verifizieren, wenn eine User-Service-Instanz neu gestartet wird, da beim Neustart ein neues Schlüsselpaar generiert wird. In der Praxis können wir daher noch nicht von einem sicher hochverfügbaren System ausgehen.

Um auf den Punkt der Agilität zu kommen, können durch Kubernetes neue Versionen von Services schnell ohne Ausfallzeit in Betrieb genommen werden. Wollen wir die Agilität hier jedoch verbessern, sollten wir eine automatisierte Test- und Deployment-Pipeline implementieren um neue Versionen mit nur einem Klick bereitzustellen.

Für die Wartbarkeit sorgt die Strukturierung in klein geschnittene Services für eine übersichtliche und gekapselte Codestruktur, die keine Auswirkungen auf andere Services hat. Änderungen könnten jedoch Fehler in den Schnittstellen zu anderen Services verursachen, die durch zusätzliche Contract-Tests abgedeckt werden könnten. Contract-Tests sind Tests die prüfen, ob die Schnittstellen, beispielsweise zwischen Frontend und Backend, bei Änderungen weiterhin kompatibel sind.

In Bezug auf die Beobachtbarkeit haben wir mithilfe unserer Monitoring-Lösung ein minimal beobachtbares System. Das Problem hier ist, dass wir Fehler oder Performanceprobleme erst zu spät erkennen, da wir weder über einen Gesundheitsstatus des Systems noch aktiv über auftretende Fehler benachrichtig werden. Hierfür müsste ein Metrik-Dashboard des Systems mit einer Alarmbenachrichtigung eingerichtet werden, welche im Ausblick weiter ausgeführt wird.

Ein weiterer wichtiger Aspekt, der bisher nicht angesprochen wurde, sind die Kosten des Systems. Die umfangreiche Nutzung verwalteter Google Cloud-Komponenten kann erheblich teurer sein als die Nutzung einer SaaS-Lösung. Möchte man wirtschaftlich mit der Konkurrenz mithalten, sollten insbesondere kostenintensive Komponenten selbst bereitgestellt und verwaltet werden. Dies würde jedoch die Komplexität des Systems weiter erhöhen und zusätzliche Zeit in Anspruch nehmen, um die Qualitätsanforderungen zu erfüllen.

5.2 Kritik an der Microservice-Architektur

Auch wenn bislang nur die Vorteile der architektonischen Ausrichtung auf Microservices betont wurden, ist es wichtig, sie kritisch zu hinterfragen, da sie in vielen Fällen möglicherweise nicht die optimale Lösung für Probleme innerhalb der IT-Landschaft darstellen.

Betrachten wir beispielsweise das Problem der Skalierbarkeit, so geht es im Grunde darum eine hohe Last an gleichzeitigen Anfragen zu bearbeiten. Gemäß Uwe Friedrichsen wäre dies schon mit einem LAMP-Stack-Server oder 10 LAMP-Stack-Serverknoten und Load Balancer ohne Microservices möglich. Seiner Einschätzung nach könnten mit dieser Konfiguration zwischen 300.000 und 3 Millionen parallele Anfragen bearbeitet werden. [Fri20b]

Ein weiteres Problem betrifft die Simplizität und das modulare Design. Es ist wichtig anzumerken, dass Microservices allein keine Garantie für ein einfacheres Codeverständnis oder ein modulares Design bieten. Die Entscheidung für eine bestimmte Architektur hat nämlich keinen unmittelbaren Einfluss auf den Quellcode einer Anwendung, was bedeutet, dass das zugrunde liegende und zu lösende Problem einer Anwendung nicht vereinfacht wird. Im Gegenteil wird durch die Einführung von Microservices die strukturelle Komplexität erhöht und potenzielle Fehlerquellen in der Anwendung verstärkt, da eine verteilte Architektur aufgrund der unvorhersehbaren Natur eines Netzwerks eine nicht deterministische Ausführung mit sich bringt. Dies bedeutet konkret, dass alle möglichen Aufrufe zwischen Services funktionieren können, aber nicht müssen. [Fri20c]

Nach Uwe Friedrichsen sind Microservices eher dann relevant, wenn extrem schnelle Deployments erforderlich sind und mehrere Teams innerhalb eines Unternehmens an einem Projekt arbeiten. Ein weiterer Grund für Microservices sind unterschiedliche nichtfunktionale Anforderungen innerhalb verschiedener Funktionen einer Anwendung. Wenn die genannten Voraussetzungen nicht erfüllt sind und kein exponentielles Wachstum des Unternehmens vorliegt, sollten alternative Architekturstile in Betracht gezogen werden: [Fri21b]

Ein Modulith ist, wie zuvor erwähnt, eine Form des Monolithen, der in modularer Weise strukturiert ist und strikten Designprinzipien folgt. [Fri21c] Dies ermöglicht die Entwicklung eines gut organisierten und leicht wartbaren Codes, der auch langfristig Bestand hat.

Ein Microlith ist ein Microservice, der bestimmte Einschränkungen hat: Er darf keine externen Abhängigkeiten für eingehende Anfragen haben und benötigt daher einen Mechanismus, um die Konsistenz der Daten zwischen den verschiedenen Microservices sicherzustellen. [Fri21a] Anders ausgedrückt bedeutet dies, dass selbst bei einem Fehlschlagen dieses Mechanismus die eingehenden Anfragen an den Service davon unberührt bleiben.

Angesichts dieses neuen Wissens können wir nun unsere Lösung anhand der nicht-funktionalen Anforderungen kritisch prüfen und zu den folgenden Schlussfolgerungen gelangen:

Für die Skalierbarkeit und Hochverfügbarkeit hätte die Bereitstellung der Anwendung mit verschiedenen modulithischen Serverknoten und Load Balancern ausgereicht. Dies hätte die Komplexität verringert, da wir uns das Definieren separater Dockerfiles und Kubernetes-Deployments für jeden Service hätten sparen können. Ein Nachrichtenbroker wie Kafka oder REST-API-Calls zwischen den Services wären somit überflüssig gewesen. Die gesamte Komplexität der Bereitstellung, der Inter-Service-Kommunikation und des Stub-Testens wäre weggefallen, was das Projekt erheblich vereinfacht hätte. Hinsichtlich der Agilität hätte Kubernetes mit modulithischen Serverknoten und einer gut konfigurierten Deployment-Pipeline schnelle Updates ohne Ausfallzeiten liefern können. Da das Projekt von einer Person bearbeitet wurde gibt es keine Vorteile hinsichtlich einer parallelen Arbeit von Teams. Es ist auch wichtig anzumerken, dass kein besonderes Augenmerk auf die Fehlerbehandlung zwischen den Microservices gelegt wurde. Wenn dies berücksichtigt worden wäre, wären komplexere Fehlerbehandlungen der Services nötig gewesen, um nicht deterministisches Verhalten zu berücksichtigen. Im Fall eines Modulithen würde diese zusätzliche Arbeit wegfallen. Was die Wartbarkeit angeht, sollte ein sehr gut geschriebener Modulith diese Anforderung erfüllen. Bezüglich der Beobachtbarkeit, fungiert der Modulith auch wie ein Log-Aggregator, da alle Logs an einer Stelle vereint sind. Dennoch könnte ein ELK-Stack hilfreich sein, um das Durchsuchen der Log-Einträge zu ermöglichen und gegebenenfalls weitere Logs von Replikaten der Anwendung zu aggregieren.

Insgesamt können wir also sagen, dass ein Modulith mit verschiedenen Replikationen in Kubernetes nach unseren Qualitätsanforderungen eine genauso so gute und weniger komplexe Lösung geboten hätte. Nach den Argumenten von Uwe Friedrichsen sind die Voraussetzungen, welche eine Microservice-Architektur begünstigen, für uns sowieso nicht erfüllt gewesen.

Trotzdem sollte überprüft werden, ob bestimmte Funktionen wie zum Beispiel die Warenkorbfunktion erheblich stärker beansprucht werden als andere. In einem solchen Fall wäre es sinnvoll, die Warenkorbfunktion in einen eigenen Service auszulagern und separat zu skalieren, um Kosten zu sparen.

5.3 Fazit

In dieser Arbeit wurde ein Prototyp einer E-Commerce-Anwendung vorgestellt, der für den Verkauf von Musikproduktionstemplates ausgelegt ist. Dieser Prototyp wurde in einer Microservice-Architektur entworfen, die sehr gut für dynamische Skalierungen und Hochverfügbarkeit geeignet ist. Jedoch konnten Vorteile wie unabhängiges paralleles Arbeiten von Teams und in diesem Zusammenhang extrem schnelle Releasezyklen im Rahmen dieses Ein-Mann-Projekts nicht ausgenutzt werden. Zudem ist wichtig zu wissen, dass diese Architektur mit einer hohen strukturellen Komplexität und hohen Fehleranfälligkeit einhergeht. Beispielsweise erzeugen zahlreiche Deployment-Dateien, verteilte Transaktionen, Kommunikations- und Überwachungskomponenten sowie Resilienzmechanismen eine zusätzliche Komplexität, die behandelt werden muss. Wählt man diese Architekturform, sollte das Team daher in der Lage sein, mit diesen Herausforderungen umzugehen.

Da die besonderen Vorteile der Microservice-Architektur in der Regel vor allem bei sehr großen Unternehmen wie Netflix oder Amazon zur Geltung kommen, wäre es für kleinere oder mittlere Unternehmen ratsamer, eine Architektur wie den Modulithen oder Microlithen zu wählen. Bei Microlithen ist zusätzlich zu beachten, dass ein nötiger Abgleichmechanismus zusätzliche Komplexität bedeutet, die erarbeitet werden muss. Zum Schluss kann man sagen, dass die Notwendigkeit von sehr gutem Design für egal welchen Architekturstil man sich entscheidet, gleich bleibt.

6 Ausblick

Während der Prototyp funktionsfähig ist und die grundlegenden Anforderungen erfüllt, fehlen ihm noch einige Aspekte und Funktionen, um im Produktionsbetrieb verwendet zu werden. Im Folgenden werden diese Punkte erläutert und weitere Optimierungen vorgeschlagen.

6.1 Validierung und Simulationsbehebung

Benutzerspezifische Fehlermeldungen sollten im Frontend angezeigt werden, um die Benutzerfreundlichkeit der Anwendung zu erhöhen. Zudem sollten alle Daten die im Frontend eingegeben werden einer gründlichen Input-Validierung unterzogen werden, um XSS-Attacken und andere Sicherheitsrisiken zu minimieren. Der E-Mail-Service sollte durch einen etablierten E-Mail-Anbieter wie MailJet oder SendGrid ersetzt werden. Ebenso ist es nötig, den Payment-Service durch einen bewährten Payment-Provider wie PayPal oder Stripe zu ersetzen. Um den Prozess noch weiter zu vereinfachen, kann ein Merchant of Record wie Fastspring eingebunden werden. Dieser übernimmt zusätzlich die Rolle des Verkäufers und kümmert sich um alle steuerrechtlichen Vorgänge, die beim Verkauf anfallen. Dadurch entfiele auch die Notwendigkeit einen eigenen Checkout-Prozess zu implementieren, da diese Aufgabe von Fastspring übernommen wird. Andernfalls müsste der Checkout-Prozess entsprechend mit dem ausgewählten Payment-Provider konfiguriert werden.

Des Weiteren sollte eine Uploadfunktion für die eigentlichen Projektdateien implementiert werden. Diese Dateien können dann über einen Downloadlink durch den E-Mail-Anbieter an den Kunden weitergeleitet werden.

6.2 Sicherheit

Statt einen eigenen Authentifizierungs- und Autorisierungsservice zu implementieren, empfiehlt es sich, OAuth 2.0 für die Autorisierung zu verwenden. OAuth 2.0 ist ein Industriestandard für die sichere Übertragung von Zugriffsrechten an Drittanbieteranwendungen. Es wird von vielen Diensten unterstützt und folgt etablierten Sicherheitsstandards. Zum Beispiel ermöglicht OAuth 2.0 die Verwendung kurzlebiger Access-Tokens zur Autorisierung, die über einen Endpunkt widerrufen werden können, um zu verhindern, dass kompromittierte Tokens im Umlauf bleiben. Zudem können durch Refresh-Tokens die kurzlebigen Access-Tokens erneuert werden. Während man den Autorisierungsserver so anpassen könnte, dass er Access-Tokens auch zur Authentifizierung nutzt, ist es sicherer, ein Authentifizierungsprotokoll wie OpenID Connect zu verwenden. OpenID Connect baut auf OAuth 2.0 auf und ermöglicht eine sichere Benutzerauthentifizierung über ein ID-Token. Dieses Token enthält Informationen über die Authentifizierungsanfrage und kann zusätzliche Benutzerinformationen bereitstellen, die über den OpenID-Connect-Endpunkt abgerufen werden können. Zur Umsetzung dieser Standards könnte eine Implementierung mit Keycloak erfolgen. Keycloak ist eine Open-Source-Plattform für Identity- und Access-Management (IAM), die sowohl OAuth 2.0 als auch OpenID Connect unterstützt. Sie bietet eine zentrale Verwaltung von Benutzern und Rollen sowie zusätzliche Sicherheitsfunktionen wie Multi-Faktor-Authentifizierung. Der selbst implementierte User- und Auth-Service aus der ersten Version des Prototyps würde damit überflüssig. Ein separater User-Service wäre nur dann erforderlich, wenn zusätzliche Benutzerfunktionen benötigt werden, die Keycloak nicht bietet.

Zudem sollte ein vordefiniertes Konto mit Admin-Rechten erstellt werden. Zurzeit wird das Admin-Konto einfach mit dem Benutzernamen *admin* erstellt.

Schließlich wurde die rechtliche Absicherung der E-Commerce-Anwendung bisher noch nicht berücksichtigt. In Zusammenarbeit mit IT-Recht-Spezialisten sollten die rechtlichen Anforderungen eines Online-Shops geprüft und die Anwendung entsprechend angepasst werden. Dies betrifft für die Entwicklung beispielsweise den Umgang mit personenbezogenen Daten oder den Einsatz von Cookies.

Obwohl ein Produktionsbetrieb mit diesen Änderungen möglich wäre, gibt es noch einige Optimierungsmöglichkeiten, die insbesondere für eine Microservice-Architektur von Bedeutung sind:

6.3 Resilienz

In verteilten Systemen besteht stets das Risiko von Teilausfällen, wenn Services synchron miteinander kommunizieren. Ein Service könnte aufgrund von Ausfällen, Wartungsarbeiten oder Überlastung nicht rechtzeitig auf Anfragen reagieren, was dazu führen kann, dass Clients blockiert werden und der Ausfall sich auf das gesamte System erstreckt. Ein Beispiel hierfür wäre, wenn das API Gateway unseres Systems längere Zeit auf die Antwort eines Services wartet und dadurch für andere Nutzer nicht verfügbar ist. [Ric18, Kapitel 3.2.3] Um solche Szenarien zu vermeiden ist es notwendig zusätzliche Maßnahmen in APIs zu implementieren, die von solchen Teilausfällen betroffen sein könnten.

Ein Vorschlag wäre das Circuit-Breaker-Muster, welches die Erfolgs-und Fehlerraten von Anfragen überprüft und sich wie ein Schalter beim Überschreiten einer definierten Fehlerquote schließt und keine weiteren Anfragen mehr akzeptiert. Erst nach einer gewissen Zeitspanne wird erneut überprüft, ob der Service verfügbar ist und bei Erfolg der Schalter wieder umgelegt sowie Anfragen zugelassen. [Ric18, Kapitel 3.2.3] Implementieren könnte man dies beispielsweise mit Resilience4j, das auch in Spring Cloud unterstützt wird.

6.4 Monitoring und Health-Checks

Wie bereits in einem früheren Teil dieser Arbeit besprochen, könnten zusätzliche Überwachungsmöglichkeiten wie Tracing und Metrik-Monitoring ergänzt werden. Für die Health-Checks kann Kubernetes durch eine *readiness probe* konfiguriert werden, um zu entscheiden, ob Anfragen bei schlechtem Gesundheitszustand an eine andere Instanz einer Anwendung geroutet werden sollen. Eine *liveness probe* wiederum entscheidet, ob Kubernetes die Instanz komplett neu starten soll. [Ric18, Kapitel 12.4.2] Ein bekanntes Beispiel für eine Health-Check-Bibliothek ist der zuvor erwähnte Spring Boot Actuator, der mithilfe seines Endpunkts eine Reihe von Gesundheitsprüfungen durchführt, die auf der verwendeten Infrastruktur der Anwendung basieren.

Literatur

[bae23] Baeldung. *Creating Kafka Topic With Docker Compose | Baeldung on Ops*. https://www.baeldung.com/ops/kafka-new-topic-docker-compose. Version 11.2023. Zugriffsdatum: 16.08.2024.

[Bas24a] Nick Basile. *Building A Comments System With Vue.js, Laravel, and Tailwind CSS Part I*. https://nickjbasile.medium.com/building-a-comments-system-with-vue-js-laravel-and-tailwind-css-part-i-e24e8518ee3. Version 04.2018. Zugriffsdatum: 27.03.2024.

[Bau+11] Christian Baun u. a. *Cloud Computing - Web-basierte dynamische IT-Services*. Springer Berlin, Heidelberg, 2011. DOI: 10.1007/978-3-642-18436-9.

[Boo] Bootstrap. *Checkout example for Bootstrap — getbootstrap.com*. https://getbootstrap.com/docs/4.0/examples/checkout/. Zugriffsdatum: 22.04.2024.

[cas19] cassiomolin. *log-aggregation-spring-boot-elastic-stack/filebeat/filebeat.docker.yml at master · cassiomolin/log-aggregation-spring-boot-elastic-stack*. https://github.com/cassiomolin/log-aggregation-spring-boot-elastic-stack/blob/master/filebeat/filebeat.docker.yml. Version 2019. Zugriffsdatum: 02.04.2024.

[cod] codeply. *Bootstrap Checkout Example Code*. https://www.codeply.com/p?starter=Bootstrap&ex=Sh3KmpOVTc. Zugriffsdatum: 22.04.2024.

[Dei22] Fabian Deitelhoff. *Vue.js - Von Grundlagen bis Best Practices*. ger. dpunkt.verlag, 2022. ISBN: 9783969107607. URL: https://content-select.com/de/portal/media/view/62145ecc-9dd0-4faf-acdb-0a29b0dd2d03.

[Ebe22] Christof Ebert. *Systematisches Requirements Engineering*. ger. 7. Aufl. dpunkt.verlag, 2022. ISBN: 9783969107683. URL: https://content-select.com/de/portal/media/view/62145eca-c094-43a8-a875-0a29b0dd2d03.

[Ela] Elastic. *Running Filebeat on Kubernetes | Filebeat Reference [7.2] | Elastic*. https://www.elastic.co/guide/en/beats/filebeat/7.2/running-on-kubernetes.html. Zugriffsdatum: 16.04.2024.

[Fri20a] Uwe Friedrichsen. *The microservices fallacy - Part 1*. https://www.ufried.com/blog/microservices_fallacy_1/. Version 11.2020. Zugriffsdatum: 15.06.2024.

[Fri20b] Uwe Friedrichsen. *The microservices fallacy - Part 2*. https://www.ufried.com/blog/microservices_fallacy_2_scalability/. Version 11.2020. Zugriffsdatum: 15.06.2024.

[Fri20c] Uwe Friedrichsen. *The microservices fallacy - Part 3*. https://www.ufried.com/blog/microservices_fallacy_3_simplicity/. Version 11.2020. Zugriffsdatum: 15.06.2024.

[Fri21a] Uwe Friedrichsen. *The microservices fallacy - Part 10*. https://www.ufried.com/blog/microservices_fallacy_10_microliths/. Version 01.2021. Zugriffsdatum: 15.06.2024.

[Fri21b] Uwe Friedrichsen. *The microservices fallacy - Part 7*. https://www.ufried.com/blog/microservices_fallacy_7_actual_reasons/. Version 01.2021. Zugriffsdatum: 15.06.2024.

[Fri21c] Uwe Friedrichsen. *The microservices fallacy - Part 9*. https://www.ufried.com/blog/microservices_fallacy_9_moduliths/. Version 01.2021. Zugriffsdatum: 15.06.2024.

[Gol20] Wolfgang Golubski. *Entwicklung verteilter Anwendungen - Mit Spring Boot & Co*. Springer Vieweg Wiesbaden, 2020. DOI: 10.1007/978-3-658-26814-5.

[HS17] Wilhelm Hasselbring und Guido Steinacker. "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, S. 243–246. DOI: 10.1109/ICSAW.2017.11.

[IBM] IBM. *Was ist DevOps?* https://www.ibm.com/de-de/topics/devops. Zugriffsdatum: 03.06.2024.

[Kaf] Kafka. *Apache Kafka*. https://kafka.apache.org/intro. Zugriffsdatum: 02.08.2024.

[KM23] Michael Kaufmann und Andreas Meier. *SQL- & NoSQL-Datenbanken*. 9. erweiterte und aktualisierte Auflage. Springer Vieweg Berlin, Heidelberg, 2023. DOI: 10.1007/978-3-662-67092-7.

[Kub24a] Kubernetes. *Deployments*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Version 03.2024. Zugriffsdatum: 10.06.2024.

[Kub24b] Kubernetes. *Ingress*. https://kubernetes.io/docs/concepts/services-networking/ingress/. Version 04.2024. Zugriffsdatum: 10.06.2024.

[Kub24c] Kubernetes. *Kubernetes Components*. https://kubernetes.io/docs/concepts/overview/components/. Version 05.2024. Zugriffsdatum: 10.06.2024.

[Kub24d] Kubernetes. *ConfigMaps*. https://kubernetes.io/docs/concepts/configuration/configmap/. Version 03.2024. Zugriffsdatum: 10.06.2024.

[Kub24e] Kubernetes. *Secrets*. https://kubernetes.io/docs/concepts/configuration/secret/. Version 07.2024. Zugriffsdatum: 26.07.2024.

[Kub24f] Kubernetes. *Service*. https://kubernetes.io/docs/concepts/services-networking/service/. Version 06.2024. Zugriffsdatum: 26.06.2024.

[Lee23] C. Lee. *How can I validate an email address in JavaScript?* https://stackoverflow.com/questions/46155/how-can-i-validate-an-email-address-in-javascript. Version 2023. Zugriffsdatum: 12.03.2024.

[Mar14] James Lewis Martin Fowler. *Microservices*. https://martinfowler.com/articles/microservices.html. Version 03.2014. Zugriffsdatum: 10.06.2024.

[MDB] MDB. *Bootstrap Shopping Carts free examples, templates & tutorial*. https://mdbootstrap.com/docs/standard/extended/shopping-carts/. Zugriffsdatum: 20.04.2024.

[New21] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. 2nd Edition. OReilly Media, Inc., 2021. ISBN: 978-1-492-03397-4.

[Pan24] Abhinav Pandey. *Spring Boot – Testing Redis With Testcontainers | Baeldung*. https://www.baeldung.com/spring-boot-redis-testcontainers. Version 01.2024. Zugriffsdatum: 15.04.2024.

[PR15] Klaus Pohl und Chris Rupp. *Basiswissen Requirements Engineering : Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level*. 4., überarbeitete Auflage. Heidelberg: dpunkt.verlag, 2015.

[Rag21] Ragvah. *Regex for checking if a string is strictly alphanumeric*. https://stackoverflow.com/questions/11241690/regex-for-checking-if-a-string-is-strictly-alphanumeric. Version 2021. Zugriffsdatum: 12.03.2024.

[Rei18] Stefan Reinheimer. *Cloud Computing - Die Infrastruktur der Digitalisierung*. Springer Vieweg Wiesbaden, 2018. DOI: 10.1007/978-3-658-20967-4.

[Ric18] Chris Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018. ISBN: 9781617294549.

[sar19] sarulabs. *Sending Docker Logs to ElasticSearch and Kibana with FileBeat - Sarulabs*. https://www.sarulabs.com/post/5/2019-08-12/sending-docker-logs-to-elasticsearch-and-kibana-with-filebeat.html. Version 08.2019. Zugriffsdatum: 28.03.2024.

[SH11] Gernot Starke und Peter Hruschka. *Software-Architektur kompakt - angemessen und zielorientiert*. 2. Aufl. Spektrum Akademischer Verlag Heidelberg, 2011. DOI: 10.1007/978-3-8274-2835-6.

[Som18] Ian Sommerville. *Software Engineering*. 10., aktualisierte Auflage. Pearson Deutschland, 2018. ISBN: 9783868943443. URL: https://elibrary.pearson.de/book/99.150005/9783863268350.

[Sta17] Stack Overflow. *How to delete the directory through java?* https://stackoverflow.com/questions/42929971/how-to-delete-the-directory-through-java. Version 2017. Zugriffsdatum: 28.02.2024.

[Ste19] Ralph Steyer. *Webanwendungen erstellen mit Vue.js - MVVM-Muster für konventionelle und Single-Page-Webseiten*. Springer Vieweg Wiesbaden, 2019. DOI: 10.1007/978-3-658-27170-1.

[Tre21] Hansruedi Tremp. *Architekturen Verteilter Softwaresysteme - SOA & Microservices - Mehrschichtenarchitekturen - Anwendungsintegration*. Springer Vieweg Wiesbaden, 2021. DOI: 10.1007/978-3-658-33179-5.

[Wel24] Kevin Welter. *Kubernetes - Das Praxisbuch für Entwickler und DevOps-Teams*. Rheinwerk Publishing Inc., 2024. URL: https://ebookcentral.proquest.com/lib/hawhamburg-ebooks/detail.action?docID=31318121.

A Anhang

A.1 Spezifikation der Use Cases

Anwendungsfall	Registrieren (uc/1)
Akteure	Gast
Vorbedingung	• Der Gast ist nicht registriert. • Der Gast befindet sich auf der Registrierungsseite.
Hauptszenario	1. Der Gast gibt die erforderlichen Informationen ein (z. B. Benutzername, Passwort, Vor- und Nachname). 2. Der Gast sendet das Registrierungsformular ab. 3. Eine Erfolgsnachricht wird angezeigt. 4. Der Benutzer wird zur Login-Seite weitergeleitet.
Alternativ-szenarien	2a. Der angegebene Benutzername existiert bereits. 2a1. Es wird eine Fehlermeldung angezeigt und der Benutzer aufgefordert, einen anderen Benutzernamen zu wählen.
Nachbedingung	• Der Gast ist nun ein registrierter Benutzer und befindet sich auf der Login-Seite. • Der registrierte Benutzer wurde in der Datenbank angelegt.

Tabelle A.1: Registrieren (uc/1)

Anwendungsfall	Anmelden (uc/2)
Akteure	Gast
Vorbedingung	• Der Gast ist ein registrierter Benutzer. • Der Gast ist nicht eingeloggt. • Der Gast befindet sich auf der Anmeldeseite.
Hauptszenario	1. Der Gast gibt Benutzernamen und Passwort ein. 2. Der Gast sendet das Anmeldeformular ab. 3. Eine Erfolgsnachricht wird angezeigt. 4. Der Benutzer wird zur Homepage weitergeleitet.
Alternativ-szenarien	2a. Die angegebenen Anmeldeinformationen sind nicht korrekt. 2a1. Es wird eine Fehlermeldung angezeigt, dass die angegebenen Anmeldeinformationen nicht korrekt sind.
Nachbedingung	• Der Gast ist im System angemeldet und wurde auf die Homepage weitergeleitet. • Der Benutzername des Benutzers wird neben dem User-Icon angezeigt. • Ein Json-Web-Token wurde für den angemeldeten Benutzer erstellt.

Tabelle A.2: Anmelden (uc/2)

Anwendungsfall	Template-Produktseite besuchen (uc/3)
Akteure	Gast, User, Admin
Vorbedingung	Keine
Hauptszenario	1. Der Akteur klickt auf ein Bild vom Template auf der Startseite. 2. Das System navigiert zur Template-Produktseite.
Alternativ-szenarien	Keine
Nachbedingung	• Der Akteur befindet sich auf der Produktseite für das ausgewählte Template.

Tabelle A.3: Template-Produktseite besuchen (uc/3)

Anwendungsfall	Tracks eines Templates abspielen und stoppen (uc/4)
Akteure	Gast, User, Admin
Vorbedingung	• Der Akteur befindet sich auf der Template-Produktseite.
Hauptszenario	1. Der Akteur klickt auf eine bestimmte zeitliche Stelle des Tracks. 2. Der Akteur klickt auf die Wiedergabetaste für einen Track. 3. Das System spielt den Track ab der bestimmten zeitlichen Stelle ab, wobei der Fortschritt des Tracks ersichtlich wird. 4. Der Akteur klickt auf die Stopp-Taste. 5. Das System stoppt den Track.
Alternativ-szenarien	4a. Der Akteur klickt nicht auf die Stopp-Taste. 4a1. Am Ende des Tracks stoppt das System den Track.
Alternativ-szenarien	1a. Der Akteur klickt auf keine bestimmte zeitliche Stelle des Tracks. 1a1. Weiter mit 2. (die zeitliche Stelle ist jetzt der Anfang des Tracks)
Nachbedingung	• Der ausgewählte Track spielt nicht mehr. • Die neue zeitliche Stelle des Tracks ist ersichtlich.

Tabelle A.4: Tracks eines Templates abspielen und stoppen (uc/4)

Anwendungsfall	Kommentare zu einem Track anzeigen (uc/5)
Akteure	Gast, User, Admin
Vorbedingung	• Der Akteur befindet sich auf der Template-Produktseite.
Hauptszenario	1. Der Akteur klickt auf das „alle Kommentare Anzeigen"-Icon des Tracks. 2. Das System zeigt Kommentare zu den Template-Tracks an. Das System errechnet dabei, wann der Kommentar erstellt wurde. (z. B. vor 6 Sekunden, vor einem Monat etc.)
Alternativ-szenarien	1a. Der Akteur fährt über das Icon eines Akteurs auf dem Track. 1a1. Das System zeigt per Tooltip den Kommentar eines Akteurs an.
Alternativ-szenarien	2a. Es gibt keine Kommentare für den Track. 2a1. Es wird eine Meldung ausgegeben, die darauf hinweist, dass es noch keine Kommentare gibt.
Nachbedingung	• Der Akteur sieht nun eine Anzeige zu den Kommentaren des Tracks.

Tabelle A.5: Kommentare zu einem Track anzeigen (uc/5)

Anwendungsfall	Template zum Warenkorb hinzufügen (uc/6)
Akteure	Gast, User, Admin
Vorbedingung	• Der Akteur befindet sich auf der Template-Produktseite.
Hauptszenario	1. Der Akteur klickt auf den „Add to Cart"-Button für ein Template. 2. Das System fügt das Template dem Warenkorb des Akteurs hinzu. 3. Eine Erfolgsmeldung, dass das Produkt dem Warenkorb hinzugefügt wurde, wird angezeigt.
Alternativ-szenarien	2a. Das Produkt befindet sich schon im Warenkorb 2a1. Eine Meldung, dass sich das Produkt schon im Warenkorb befindet, wird angezeigt.
Nachbedingung	• Das ausgewählte Template wurde dem Warenkorb des Akteurs hinzugefügt. • Eine Session für den Warenkorb wurde vom System erstellt. • Das ausgewählte Template wurde in der Datenbank des Warenkorbs angelegt.

Tabelle A.6: Template zum Warenkorb hinzufügen (uc/6)

Anwendungsfall	Warenkorb anzeigen (uc/7)
Akteure	Gast, User, Admin
Vorbedingung	• Im Warenkorb des Akteurs befinden sich Templates.
Hauptszenario	1. Der Akteur klickt auf das Warenkorb-Symbol. 2. Das System zeigt den Inhalt des Warenkorbs des Gastes auf einer neuen Seite an.
Alternativ-szenarien	2a. Der Warenkorb ist leer. 2a1. Eine Meldung, dass der Warenkorb leer ist, wird angezeigt.
Nachbedingung	• Der Akteur befindet sich auf der Seite des Warenkorbs. • Der Inhalt des Warenkorbs wird angezeigt.

Tabelle A.7: Warenkorb anzeigen (uc/7)

Anwendungsfall	Template aus dem Warenkorb löschen (uc/8)
Akteure	Gast, User, Admin
Vorbedingung	• Der Akteur sieht sich den Warenkorb auf der Cartpage an. • Der Warenkorb ist nicht leer.
Hauptszenario	1. Der Akteur klickt auf das „Delete“-Icon neben einem Template im Warenkorb. 2. Das System entfernt das ausgewählte Template aus dem Warenkorb.
Alternativ-szenarien	Keine
Nachbedingung	• Das ausgewählte Template wurde aus dem Warenkorb des Akteurs entfernt. • Das ausgewählte Template wurde aus der Datenbank des Warenkorbs entfernt.

Tabelle A.8: Template aus dem Warenkorb löschen (uc/8)

Anwendungsfall	Template kaufen (uc/9)
Akteure	Gast, User, Admin
Vorbedingung	• Im Warenkorb des Akteurs befinden sich Templates. • Der Akteur befindet sich auf der Seite des Warenkorbs.
Hauptszenario	1. Der Akteur klickt auf den „Checkout“-Button. 2. Das System leitet den Akteur auf die Checkoutpage weiter. 3. Der Akteur gibt Zahlungs- und Versanddetails ein. 4. Der Akteur klickt auf den „Buy Now“-Button. 5. Das System verarbeitet die Zahlung. 6. Nach Erfolgreicher Verarbeitung erstellt das System einen Downloadlink. 7. Eine Erfolgsmeldung wird angezeigt. 8. Der Akteur wird zur Homepage weitergeleitet. 9. Der Warenkorb wird geleert.
Alternativ-szenarien	4a. Die Zahlungs- oder Versanddetails weisen Fehler auf. 4a1. Eine Fehlermeldung und Aufforderung korrekte Daten einzugeben wird angezeigt. 4a2. Weiter bei 3
Alternativ-szenarien	4a. Das Konto ist nicht ausreichend für die Zahlung gefüllt. 4a1. Eine Fehlermeldung wird angezeigt. (In der Konsole)
Alternativ-szenarien	4a. Der Warenkorb ist leer. 4a1. Eine Fehlermeldung wird angezeigt. 4a2. Der Akteur fügt ein Produkt zum Warenkorb hinzu. 4a3. Weiter bei 1.
Nachbedingung	• Die ausgewählten Templates wurden erfolgreich vom Akteur gekauft. • Die Bestellung und die Transaktion wurden in der Datenbank gespeichert. • Die Bestellung wurde in Elasticsearch indexiert. • Die gekauften Produkte wurden aus der Datenbank des Warenkorbs gelöscht.

Tabelle A.9: Template kaufen (uc/9)

Anwendungsfall	Kommentieren von Tracks eines Templates (uc/10)
Akteure	User, Admin
Vorbedingung	• Der Gast ist angemeldet und hat die Rolle User oder Admin. • Der Akteur befindet sich auf der Template-Produktseite
Hauptszenario	1. Der Akteur klickt auf das „Kommentieren“-Icon eines Tracks. 2. Der Akteur gibt einen Kommentar im Kommentarbereich ein. 3. Der Akteur sendet den Kommentar mittels „Publish“-Button ab. 4. Das System speichert den Kommentar ab und zeigt die Initialen des Akteurs als Icon relativ zum Timestamp des erstellten Kommentars an.

Tabelle A.10: Kommentieren von Tracks eines Templates (uc/10)

Anwendungsfall	Eigene Kommentare zu einem Track löschen (uc/11)
Akteure	User, Admin
Vorbedingung	• Ein Kommentar des Akteurs ist am Track vorhanden. • Der Akteur ist angemeldet. • Der Akteur befindet sich auf der Produktseite des Templates
Hauptszenario	1. Der Akteur klickt auf das „Kommentare Anzeigen“-Icon eines Tracks. 2. Der Akteur findet seinen eigenen Kommentar zu einem Track. 3. Der Akteur klickt auf das Bearbeiten Icon neben seinem Kommentar. 4. Der Akteur klickt auf den angezeigten „Delete“-Button. 5. Das System löscht den Kommentar des Akteurs.
Alternativ-szenarien	Keine
Nachbedingung	• Der Kommentar des Akteurs wurde aus der Datenbank entfernt.

Tabelle A.11: Eigene Kommentare zu einem Track löschen (uc/11)

Anwendungsfall	Eigene Kommentare zu einem Track bearbeiten (uc/12)
Akteure	User, Admin
Vorbedingung	• Ein Kommentar des Akteurs ist am Track vorhanden. • Der Akteur ist angemeldet. • Der Akteur befindet sich auf der Produktseite des Templates
Hauptszenario	1. Der Akteur klickt auf das „Kommentare Anzeigen“-Icon eines Tracks. 2. Der Akteur findet seinen eigenen Kommentar zu einem Track. 3. Der Akteur klickt auf das „Bearbeiten“-Icon neben seinem Kommentar. 4. System zeigt den Kommentar in einer bearbeitbaren Form an. 5. Der Akteur modifiziert den Kommentar. 6. Der Akteur klickt den „Update“-Button des Kommentars. 7. Das System speichert den neuen Kommentar.
Alternativ-szenarien	4a. Der Akteur klickt auf den „Cancel“-Button. 4a1. Der originale Kommentar bleibt unverändert und wird nicht mehr in bearbeitbarer Form angezeigt.
Nachbedingung	• Der bearbeitete Kommentar wurde in der Datenbank gespeichert. • Der bearbeitete Kommentar wird in der normalen Kommentaransicht angezeigt.

Tabelle A.12: Eigene Kommentare zu einem Track bearbeiten (uc/12)

Anwendungsfall	Abmelden (uc/13)
Akteure	User, Admin
Vorbedingung	• Der Akteur ist angemeldet.
Hauptszenario	1. Der Akteur klickt auf den „Logout“-Button. 2. Das System meldet den Benutzer ab.
Alternativ-szenarien	Keine
Nachbedingung	• Der Akteur ist im System abgemeldet. • Der JSON-Web-Token wurde vom System gelöscht.

Tabelle A.13: Abmelden (uc/13)

Anwendungsfall	Eigenen Account löschen (uc/14)
Akteure	User, Admin
Vorbedingung	• Der Akteur ist angemeldet.
Hauptszenario	1. Der Akteur navigiert zum „Delete Account“-Button des Profils. 2. Der Akteur klickt auf „Delete Account“. 3. Das System löscht den Account des Akteurs. 4. Eine Erfolgsmeldung wird angezeigt.
Alternativ-szenarien	Keine
Nachbedingung	• Der Akteur ist im System abgemeldet. • Der JSON-Web-Token wurde vom System gelöscht. • Der Akteur wurde aus der Datenbank gelöscht.

Tabelle A.14: Eigenen Account löschen (uc/14)

Anwendungsfall	Template aus dem Shop löschen (uc/15)
Akteure	Admin
Vorbedingung	• Der Admin ist angemeldet. • Das Template existiert im Shop. • Der Admin befindet sich auf der Homepage.
Hauptszenario	1. Der Admin findet das zu löschende Template. 2. Der Admin klickt auf das „Delete“-Icon des Templates. 3. Das System löscht das Template und alle zugehörigen Kommentare der Tracks. 4. Das System zeigt eine Erfolgsmeldung an.
Alternativ-szenarien	Keine
Nachbedingung	• Das ausgewählte Template wurde aus der Datenbank gelöscht. • Alle zugehörigen Tracks des Templates wurden aus der Datenbank gelöscht. • Alle Kommentare der Tracks des Templates wurden aus der Datenbank gelöscht.

Tabelle A.15: Template aus dem Shop löschen (uc/15)

Anwendungsfall	Produktinformationen eines Templates bearbeiten (uc/16)
Akteure	Admin
Vorbedingung	• Der Admin ist angemeldet. • Das Template existiert im Shop. • Der Admin befindet sich auf der Produktseite.
Hauptszenario	1. Der Admin navigiert zum Admin-Dashboard. 2. Der Admin findet das zu bearbeitende Template. 3. Der Admin klickt das „Bearbeiten“-Icon auf der Produktseite an. 4. Das System zeigt die aktuellen Informationen in einer bearbeitbaren Form an. 5. Der Admin modifiziert die Produktinformationen. 6. Der Admin sendet die bearbeiteten Informationen ab. 7. Das System zeigt eine Erfolgsmeldung an.
Alternativ-szenarien	5a. Die Produktinformationen enthalten Fehler im Zeichensatz. 5a1. Eine Fehlermeldung wird angezeigt und der Admin gebeten die Fehler zu beheben. 5a2. Weiter bei 5.
Alternativ-szenarien	5a. Der Admin klickt auf den „Cancel“-Button und bricht die Bearbeitung ab. 5a1. Die Form wird geschlossen und die originalen Informationen bleiben unverändert.
Nachbedingung	• Die Produktinformationen für das ausgewählte Template wurden in der Datenbank gespeichert.

Tabelle A.16: Produktinformationen eines Templates bearbeiten (uc/16)

Anwendungsfall	Template zum Shop hinzufügen (uc/17)
Akteure	Admin
Vorbedingung	• Der Admin ist angemeldet.
Hauptszenario	1. Der Admin klickt auf den „Add Product“-Button. 2. Das System fragt nach den erforderlichen Produktinformationen. 3. Der Admin gibt die erforderlichen Informationen ein. 4. Der Admin schickt die Informationen über den „Submit“-Button an das System. 5. Das System legt mithilfe der Informationen und Dateien ein neues Template an. 6. Eine Erfolgsmeldung für jede einzelne Datei wird angezeigt. 7. Die Form der Produktinformationen wird wieder zurückgesetzt.
Alternativ-szenarien	3a. Der Admin gibt fehlerhafte Informationen an. (bspw. Zeichensatz) 3a1. Der Admin wird gebeten die Informationen zu korrigieren. 3a2. Weiter bei 3.
Alternativ-szenarien	3a. Der Admin gibt nicht alle benötigten Metadaten an. 3a1. Der Admin versucht über den „Submit“-Button die Template hochzuladen. 3a2. Eine entsprechende Fehlermeldung wird angezeigt und der Submit-Prozess blockiert. 3a3. Weiter bei 3
Alternativ-szenarien	3a. Der Admin lädt keine Haupt-Audiospur hoch. 3a1. Der Admin versucht über den „Submit“-Button die Template hochzuladen. 3a2. Eine entsprechende Fehlermeldung wird angezeigt und der Submit-Prozess blockiert. 3a3. Weiter bei 3
Alternativ-szenarien	4a. Beim Anlegen wurde vom Admin kein Bild hinzugefügt. 4a1. Das System benutzt ein Standardbild für das Template. 4a2. Weiter zu 4.
Nachbedingung	• Das neue Template wurde erfolgreich dem Shop hinzugefügt. • Das Template und alle erforderlichen Daten wurden in der Datenbank angelegt.

Tabelle A.17: Template zum Shop hinzufügen (uc/17)

Anwendungsfall	Alle getätigten Bestellungen anzeigen (uc/18)
Akteure	Admin
Vorbedingung	• Der Admin ist angemeldet.
Hauptszenario	1. Der Admin navigiert zum „Orders“-Button und klickt diesen. 2. Das System zeigt eine Liste aller von Käufern getätigten Bestellungen an.
Alternativ-szenarien	3a. Es sind keine Bestellungen vorhanden. 3a1. Eine Meldung wird angezeigt, die darauf hinweist, dass es noch keine Bestellungen gibt.
Alternativ-szenarien	1a. Der Admin navigiert zur Adresse von Kibana. 1a1. Der Admin schaut sich das Kibana-Dashboard zu den Bestellungen an.
Nachbedingung	• Eine Liste aller von Käufern getätigten Bestellungen wurde dargestellt.

Tabelle A.18: Alle getätigten Bestellungen anzeigen (uc/18)

Glossar

2PC (Two-Phase Commit) Ein Transaktionsprotokoll, welches die Datenkonsistenz in einem verteilten System sicherstellt. Dabei müssen alle beteiligten Akteure einer Transaktion zugestimmt haben, um sie durchzuführen.

Base64 Ein Codierungsverfahren, welches Binärdaten in Textdaten umwandelt.

Cluster Bezeichnet den Verbund mehrerer Server zu einer Einheit.

Continuous Deployment (CD) Ein Entwicklungsprozess, der auf Continuous Integration aufbaut und die Software automatisiert in die Produktionsumgebung bereitstellt.

Continuous Integration (CI) Ein Entwicklungsprozess, bei dem Codeänderungen in ein gemeinsames Repository zusammengeführt und automatisiert getestet werden.

CORS CORS (Cross-Origin Resource Sharing) ist ein Sicherheitsmechanismus, der auf HTTP-Headern basiert und es Webbrowsern ermöglicht, Ressourcen von einer anderen Ursprungsdomäne zu laden, als der, von der die Webanwendung stammt.

CSRF CSRF (Cross-Site Request Forgery) ist ein Angriff, bei dem ein bösartiger Akteur im Namen eines authentifizierten Benutzers unbemerkte Anfragen an eine Webanwendung sendet, um unerwünschte Aktionen durchzuführen.

Dependency Injection Entwurfsmuster zur Bereitstellung von Abhängigkeiten an eine Klasse von außen und zur Laufzeit.

Deployment Als Deployment wird der Prozess der Installation, Konfiguration, Bereitstellung und Wartung einer Software in der Produktionsumgebung bezeichnet.

Deployment-Pipeline Automatisierte Abfolge von Prozessen zur Bereitstellung von Software. Grundlage für Continuous Deployment.

Dirty Read Lesezugriff einer Transaktion von Daten, die von einer anderen Transaktion geändert, aber noch nicht abgeschlossen wurden.

DNS Steht für Domain Name System und definiert ein System, welches Domainnamen in IP-Adressen übersetzt.

Docker Docker ist eine Plattform zur Containerisierung von Anwendungen. Diese können in einer isolierten Umgebung erstellt und ausgeführt werden.

Docker-Image Ein Docker-Image ist ein Paket für eine Anwendung, welches alle Abhängigkeiten enthält, die diese Anwendung benötigt, um ausgeführt zu werden. Auf Basis dieses Images kann Docker die Anwendung in einem Container ausführen.

Endpunkt Ein Zugangspunkt in einem System, über den Daten gesendet und empfangen werden können.

Healthcheck Eine Methode zur Überprüfung der Gesundheit und Verfügbarkeit der Anwendung, beispielsweise über Endpunkte.

HTTP HTTP (Hypertext Transfer Protocol) ist ein Netzwerkprotokoll, das die Kommunikation und Datenübertragung zwischen Clients und Servern definiert.

HTTP-Header HTTP-Header liefern zusätzliche Informationen zu einer HTTP-Anfrage oder Antwort.

Java Eine beliebte objektorientierte Programmiersprache für verschiedenste Anwendungen.

JavaScript Eine Skriptsprache, die für die Entwicklung von Webanwendungen verwendet wird.

JSON-Web-Token (JWT) Ein standardisiertes Tokenformat auf Basis von JSON, das zur sicheren Übertragung von Authentifizierungs- und Autorisierungsinformationen zwischen Clients und Servern dient und durch eine digitale Signatur geschützt ist.

JUnit Ein Framework für das Testen von Java-Anwendungen.

Key Value Store Eine Datenbank, die Daten in Form von einzigartigen Schlüssel-Wert-Paaren speichert.

Kubernetes Kubernetes ist eine Orchestrierungsplattform für Container-Anwendungen, die Aspekte wie Betrieb und Skalierung automatisiert.

LAMP-Stack Kombination aus Linux, Apache, MySQL und PHP zur Entwicklung von Webanwendungen.

Mixing Prozess des Abmischens aller Spuren einer Musikproduktion um ein angenehmes Klangbild zu erzeugen.

Monitoring Kontinuierliche Überwachung und Analyse eines Systems.

Nachrichtenbroker Software, welche Nachrichten zwischen verschiedenen Anwendungen oder Systemen übermittelt und so asynchrone Kommunikation ermöglicht.

Public Key Ein öffentlicher Schlüssel, der in der asymmetrischen Kryptografie verwendet wird, um Daten zu verschlüsseln oder digitale Signaturen zu überprüfen.

Registry Eine Registry oder Container-Registry ist ein Speicherort für Container-Images. Docker verwendet die Registry, um die Images abzurufen und sie in Containern auszuführen.

REST REST (Representational State Transfer) ist ein Architekturstil für die Kommunikation zwischen Systemen, der auf dem HTTP-Protokoll basiert und sich auf das Ressourcenmanagement konzentriert.

SaaS Bei Saas (Software as a Service) wird eine fertige Softwareanwendung als Dienst über das Internet bereitgestellt.

Saga Ein Entwurfsmuster, welches die Datenkonsistenz in einem verteilten System sicherstellt. Dabei wird eine Operation über mehrere Anwendungen in eine Serie von kleineren reversiblen Transaktionen unterteilt.

Service Discovery Ist der Prozess, bei dem ein System automatisch alle verfügbaren Services eines Netzwerks findet.

Stack Kombination von Software-Komponenten, die zusammen eine Anwendung oder Lösung bilden.

Vue.js Ein progressives JavaScript-Framework zur Erstellung von Benutzeroberflächen und Single Page Applications.

XSS Cross-Site Scripting (XSS) ist ein Angriff bei dem schadhafter Code (meist JavaScript) beispielsweise über Formulare in Webanwendungen eingeschleust und ausgeführt wird.

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original