



**Hochschule für Technik
und Wirtschaft Berlin**
University of Applied Sciences

Entwicklung eines Portals zur Tennisspielersuche unter der Verwendung von Microservices

Bachelorarbeit

Angestrebter Abschluss

Bachelor of Science

an der

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang Angewandte Informatik

- 1. Gutachter:** Prof. Dr. Christian Herta
- 2. Gutachter:** Tobias Dumke

Eingereicht von:

Sakhr Nabil Abdulrazzaq Al-absi

Datum: 19.04.2022

Inhaltsverzeichnis

1 Einleitung	8
1.1 Motivation	8
1.2 Zielsetzung	9
1.3 Abgrenzung	9
1.4 Aufbau der Arbeit	9
2 Grundlagen	11
2.1 Miroservices-Architektur	11
2.1.1 Microservices	12
2.1.2 Microservices-Implementierung mit Python	13
2.2 Interkommunikation zwischen den Services	13
2.2.1 Service-Discovery	13
2.2.2 Service Mesh	14
2.2.3 Message Queues (MQ's)	15
2.2.4 API Gateways	15
2.2.5 RabbitMQ	16
2.3 Agile Entwicklung mit Scrums	17
2.4 Frameworks im Projektbereich	19
2.4.1 Django	19
2.4.2 ReactJS	20
2.5 Containers	22
2.5.1 Docker	24
2.5.2 Kubernetes	28
2.6 Travis-CI	29
3 Anforderungsanalyse	30
3.1 Zielsetzung	30
3.2 Anwendungsumgebung	31
3.3 Rahmenbedingungen	31
3.4 Anforderungen	32
3.4.1 Akteure	32
3.4.2 Anwendungsfälle und User Stories	33

3.4.3	Use-Case-Diagramm	39
3.4.4	MoSCoW-Methode	40
4	Systementwurf und Konzeption	42
4.1	Entwurf Systemarchitektur	43
4.2	Abgrenzung	44
4.3	Festlegung der Technologien	44
4.4	Model-Layer	45
4.4.1	Initiative 1: User-Management	46
4.4.2	Initiative 2: Review-Management	46
4.5	View-Layer	47
4.6	Controller-Layer	57
4.7	RabbitMQ	62
5	Implementierung	66
5.1	Systemarchitektur	66
5.2	Technologien	68
5.2.1	Django	68
5.2.2	Docker	69
5.2.3	RabbitMQ	73
5.2.4	NGINX	78
5.3	Model-Layer	81
5.4	Controller-Layer	88
5.5	View-Layer	95
6	Testing	98
6.1	Unit-Tests	98
6.2	Integration-Tests	99
6.3	Test-Coverage	100
6.4	Continuous-Integration	101
7	Code-Dokumentationen	102
8	Zusammenfassung und Ausblick	103
8.1	Zusammenfassung	103
8.2	Ausblick	104

Abbildungsverzeichnis

2.3.1	User Stories und Epics (Quelle: eigene Darstellung)	18
2.4.1	Skizze einer DJANGO-MVT-Struktur, die erstellt wird aus einem Javapoint-Artikel (Jayanandana 2018).	20
2.4.2	ReactJS Bekanntheitsgrad von 2015 bis 2020 aus (Ram 2021)	21
2.5.1	Skizze der Infrastruktur eines Containers aus einem Backblaze-Artikel (Clany 2021)	23
2.5.2	Skizze eines anhand eines Mediumartikels erstellten Docker-Images. Es wird in eine Datei namens Dockerfile geschrieben und in ein Docker-Image übersetzt, bevor aus dem Image ein Container erstellt wird. (Jayanandana 2018)	26
2.5.3	Befehle im Dockerfile nach dem Artikel von Ionos (KnowHow 2021)	27
3.4.1	User Stories und Epics. (Quelle: eigene Darstellung)	34
3.4.2	User-Case-Diagram	39
4.1.1	Grobe Übersicht des gesamten Architektur (Quelle: eigene Darstellung)	43
4.5.1	Mockup – Registrierungsfenster	48
4.5.2	Mockup – Einloggen eines Benutzers	49
4.5.3	Mockup - Hauptseite und der Suche nach Benutzern	50
4.5.4	Auswahlliste des Benutzers entsprechend dem Anmeldungsstatus	51
4.5.5	Mockup - Profilseite eines Benutzers	52
4.5.6	Mockup - Meine Profilseite	53
4.5.7	Mockup - Admin Dashboard und alle registrierten Benutzern	54
4.5.8	Mockup - Admin Dashboard und alle Bewertungen einer Benutzer-ID	55
4.5.9	Mockup - Mockup - Admin Dashboard und die Bewertung-ID einer spezifischen Benutzer-ID	56
4.7.1	RabbitMQ - publish und subscribe für die Veröffentlichung und den Konsum von Nachrichten von dem RabbitMQ-Server (Quelle: eigene Darstellung)	62
5.1.1	Die Systemarchitektur zeigt die Verbindung zwischen den verschiedenen Containern der Services und was in jedem davon passiert (Quelle: eigene Darstellung)	67
5.2.1	Manche Info. in der Konfigurationsdateien von Django	68
5.2.2	Die Konfigurationen von den user_manage_api3 und NGINX Services in der docker-compose.yml Datei	70

5.2.3	Die Konfigurationen vom RabbitMQ-Service in der Konfiguration von den internen und externen Netzwerken in der docker-compose.yml Datei	71
5.2.4	Dockerfile von dem user_manage_api3-Container des User-Services	72
5.2.5	Dockerfiles von RabbitMQ und NGINX	73
5.2.6	Die Verbindungskonfiguration vom Producer zu RabbitMQ im User-Service	74
5.2.7	Die Veröffentlichung einer Nachricht bei Registrierung eines Benutzers im User-Service	75
5.2.8	Die Einrichtung von „DJANGO_SETTINGS_MODULE“ in consumer.py des Review-Services	76
5.2.9	Die Verbindungskonfiguration vom Producer zu RabbitMQ im Review-Service	76
5.2.10	Der Empfang verschiedener Nachrichten innerhalb der Callback-Funktion im Review-Service	77
5.2.11	Konfiguration, um Nachrichten von RabbitMQ im Review-Service zu konsumieren	78
5.2.12	Das Hinzufügen von verschiedenen Servern oder Domänen in der NGINX-Konfigurationsdatei	78
5.2.13	Die Konfiguration von Header und Hauptpfad zum User-Service	80
5.2.14	Die Hauptpfade zum Review-Service und zu den Administrationspanels der beiden Services	80
5.2.15	Die Pfade zu mediafiles und staticfiles	81
5.3.1	Entity-Relationship-Diagramm des User-Service	82
5.3.2	Der User-Model	83
5.3.3	Die Konfiguration des User-Adminpanels	84
5.3.4	Der Profile-Model	85
5.3.5	Entity-Relationship-Diagramm des Review-Service	87
5.4.1	Methode zur Registerierung eines Benutzers in das Portal	88
5.4.2	Veröffentlichung des Benutzernamens während der Erstellung eines Benutzers im User-Service	88
5.4.3	Klassen für die Aufgaben von EPIC 2	89
5.4.4	Veröffentlichung des Anmeldungsstatus nach einer erfolgreichen Anmeldung im User-Service	90
5.4.5	Veröffentlichung des Anmeldungsstatus nach einer erfolgreichen Abmeldung im User-Service	90
5.4.6	Klassen zu den Aufgaben von EPIC 3 in Initiative 1	91
5.4.7	Klassen zu den Aufgaben von EPIC 4 in Initiative 1	92
5.4.8	Klassen und Methoden zu den Aufgaben von EPIC 1 in Initiative 2	93
5.4.9	Überprüfung des Anmeldungsstatus eines Bentuzers aus dem User-Service im Review-Service	94
5.5.1	Die Erstellung eines Authorization-Header mit einem JWT Präfix im Frontend-Service	95
5.5.2	Eine Methode, um nicht autorisierte Ressourcen aus dem Backend zu erhalten	96
5.5.3	Eine Methode, um autorisierte Ressourcen aus dem Backend zu erhalten	96

5.5.4	Eine Methode zur Iteration durch abgefragte Benutzer	96
5.5.5	Überprüfungs des Authentifizierungsstatus im Frontend-Service, um berech-tigte Daten entsprechend anzuzeigen	97
5.5.6	Überprüfung des Autorisierungsstaus, um berechtige Daten entsprechend an-zuzeigen	97
6.1.1	Manche Unit-Tests im User-Service	99
6.2.1	Ein Beispiel für einen Integrationstest	99
6.3.1	Abdeckungsbericht über die abgeschlossenen Tests im User-Service	100
6.3.2	Abdeckungsbericht über die abgeschlossenen Tests im Review-Service . . .	100

Tabellenverzeichnis

3.1	Use Case Vs. User story	35
3.2	Initiative 1: Epic - Benutzerregisterierung	36
3.3	Initiative 1: Epic - Benutzeranmeldung	36
3.4	Initiative 1: Epic - Benutzerkategorien und Suche	36
3.5	Initiative 1: Epic - Profilwaltung	37
3.6	Initiative 2: Epic - CRUD von Bewertungen	37
3.7	Initiative 3: Epic - Verwaltung von freien Timeslots	38
3.8	Initiative 4: Epic - Anfragnverwaltung	38
3.9	MoSCoW Methode	41
4.1	Initiative 1 - User-Management-Service API's	58
4.2	Initiative 2 - Review-Management API's	60
4.3	Isolierte Aufgaben ohne Kommunikationsbedarf zwischen den Services	63
4.4	User-Service-Tasks, die über den RabbitMQ-Server veröffentlicht und konsumiert werden	64
4.5	Review-Service-Tasks, die über den RabbitMQ-Server veröffentlicht und konsumiert werden.	65

Kapitel 1

Einleitung

1.1 Motivation

Heutzutage ist die Entwicklung von Websites und Anwendungen ein wesentlicher Bestandteil unseres täglichen Geschäftslebens geworden. Jedes Unternehmen ist auf der Suche nach der optimalsten Art der Entwicklung, bei der es seine verfügbaren Ressourcen (Zeit, Geld, Mitarbeiter usw.) nutzen kann. Die Synchronität zwischen den Teammitgliedern, bei der jeder seine eigene Funktion hinzufügen kann, ohne die Arbeit der anderen zu beeinträchtigen, ist nahezu obligatorisch. In Verbindung mit dem Vorhandensein verschiedener Programmiersprachen kann es manchmal schwierig sein, die richtige Sprache für ein bestimmtes Feature oder Projekt zu wählen. Daher ist eine Architektur, durch die die Funktionen lose gekoppelt gehalten werden und ermöglicht wird, dass verschiedene Entwickler an verschiedenen Funktionen arbeiten, ohne die Arbeit der anderen zu beeinträchtigen. Eine Architektur, die sich die Tatsache zunutze macht, dass einige Entwickler in bestimmten Programmiersprachen besser sind als andere, was die Unabhängigkeit von „Sprachabhängigkeiten“ erhöhen kann, steigert die Effizienz der Entwicklung verschiedener unabhängiger Funktionen und verringert die Komplexität großer Projekte um ein Vielfaches.

Aus diesem Grund wurde im Mai 2012 der Begriff „Microservices“ eingeführt. Laut Pautasso u. a. 2017 in einem Artikel von IEEE Software sind Microservices eine Spezialisierung eines Implementierungsansatzes für serviceorientierte Architekturen (SOA) zum Aufbau flexibler, unabhängig einsetzbarer Softwaresysteme. Der Microservices-Ansatz ist eine erste Umsetzung von SOA, die auf die Einführung von DevOps folgte und für den Aufbau kontinuierlich bereitgestellter Systeme immer beliebter wird Farcic 2014. Durch den „Cloud Microservices Market Research Report“ prognostiziert Research 2020, dass der globale Markt für Microservice-Architekturen von 2019 bis 2026 mit einer CAGR von 21,37 wachsen und bis 2026 3,1 Milliarden US-Dollar erreichen wird. Aus diesem Grund wird in dieser Arbeit die Entwicklung eines Portals unter Verwendung der Microservice-Architektur angestrebt. Dies wird den gesamten Implementierungsprozess weniger kompliziert machen und aufgrund der

hohen losen Kopplungsfähigkeit von Microservices wird es auch für zukünftige Entwickler unkomplizierter sein, dem Projekt etwas hinzuzufügen, ohne die anderen bereits implementierten Funktionen zu beeinflussen. Sie werden auch in der Lage sein, ihre eigenen Funktionen mit den Sprachen einzubauen, die sie für geeignet halten und die sie am besten beherrschen.

1.2 Zielsetzung

Das Ziel besteht darin, eine Plattform in verschiedenen Sprachen zu entwickeln und sie sowie ihre Abhängigkeiten in verschiedene Container zu verpacken, sodass sie voneinander getrennt werden und gleichzeitig miteinander kommunizieren können. Die Kommunikation erfolgt dann über sprachneutrale Protokolle wie ‚Representational State Transfer‘ (REST) oder Messaging-Anwendungen wie ‚RabbitMQ‘. Die Idee ist, darzustellen, wie eine Website mithilfe von Microservices viel effizienter entwickelt werden kann. Die Website selbst wird dazu dienen, dass verschiedene Tennisspieler einander in Bezug auf ihre Stärke, ihr Alter, ihr Geschlecht usw. finden können. Außerdem können sich die Spieler gegenseitig Anfragen schicken und sich gegenseitig bewerten. Wenn in Zukunft weitere Funktionen von einem anderen Studenten hinzugefügt werden sollen, sollte dies aufgrund der verwendeten Architektur möglich sein.

1.3 Abgrenzung

Alle Prinzipien und Funktionalitäten werden berücksichtigt, um zu einem systematischen Ansatz zu gelangen und jedes Feature entsprechend zu entwickeln. Drei Services werden im Vordergrund dieses Projekts stehen. Zwei davon werden mit Python unter Verwendung von Django implementiert, einer zur Verwaltung der Benutzer und der andere, um Bewertungen für sie zu hinterlassen. Diese werden für den Backend-Teil verantwortlich sein. Das Frontend und der dritte Service werden mit JavaScript unter Verwendung der ReactJS-Bibliothek implementiert. Dazu werden zunächst die Anforderungen gesammelt und dann je nach Bedarf sowie Effizienz auf einen Service verteilt bzw. zugewiesen. Anschließend werden Design-Patterns und Diagramme skizziert, um ein besseres Verständnis für die Implementierungsschritte zu bekommen. Da auch die Zeit für diese Arbeit begrenzt ist, ist es bedeutsam, einen Ansatz zu wählen, durch den das Risiko minimiert wird, das Hauptziel der Arbeit nicht zu erreichen. Aus diesem Grund wird das Hauptziel mithilfe der MoSCoW-Methode in verschiedene kleine Ziele mit unterschiedlichen Prioritäten aufgeteilt. Auf diese Ziele und Prioritäten wird in Kapitel 3 näher eingegangen.

1.4 Aufbau der Arbeit

Zu Beginn der Arbeit werden die Grundlagen der einzelnen Architekturen, die für die Entwicklung des Portals zu verwendenden Technologien und die Bedeutung einer agilen Entwicklung geklärt. Daran wird direkt die Anforderungsanalyse der Anwendung angeschlossen, bei

der zunächst die Bedürfnisse und Funktionalitäten aus der Sicht des Benutzers erhoben werden. Sobald eine kurze Vorstellung davon vorliegt, welche Funktionalitäten benötigt werden, um ein zufriedenstellendes Endergebnis zu erzielen, werden die Anforderungen aus der Sicht des Entwicklers aufgelistet und auf den entsprechenden Service verteilt. Sobald feststeht, welche Funktion von welchem Service übernommen werden soll, wird anschließend mithilfe verschiedener Entwurfsmuster und Architekturstile ein Weg entworfen, wie diese Funktionen von diesen Services gehalten werden. Dies wird uns als Entwicklern einen Einblick geben, wie die Funktionen strukturiert und effizient implementiert werden können. Diesen Entwurfsmustern folgend wird dann die Implementierung des Projekts durchgeführt. Sobald die Implementierung abgeschlossen ist, wird jede Funktionalität entsprechend getestet. Auf diese Weise können wir etwaige Unregelmäßigkeiten oder Fehler im Projekt feststellen, bevor es an den Endbenutzer ausgeliefert wird. Während der Implementierung wird ein gehosteter kontinuierlicher Integrationsdienst wie ‚Travis-CI‘ verwendet, um unsere Software auf GitHub kontinuierlich zu testen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Definitionen und Klarstellungen einiger für das Projekt bedeuternder Begriffe erläutert.

2.1 Miroservices-Architektur

Das relevanteste Element ist die Fähigkeit zur unabhängigen Bereitstellung. Regel Nummer 1 für die Erstellung eines erfolgreichen Microservices-Systems ist es, sicherzustellen, dass jeder Microservice so unabhängig wie möglich vom Rest arbeiten kann. Das gilt für die Entwicklung, das Testen und die Bereitstellung. Dies ist das Schlüsselement, das die parallele Entwicklung zwischen verschiedenen Teams ermöglicht und ihnen die Möglichkeit gibt, die Arbeit zu skalieren. Die Geschwindigkeit von Änderungen in einem komplexen System wird dadurch erhöht.

Laut Buelta 2019 steht die Microservice-Architektur in engem Zusammenhang mit den Grundsätzen der kontinuierlichen Integration und der kontinuierlichen Bereitstellung (Continuous Deployment). Kleine Services lassen sich leicht auf dem neuesten Stand halten und kontinuierlich aufbauen sowie ohne Unterbrechung bereitstellen. Theoretisch ist es zwar möglich, völlig unverbundene Services zu haben, aber in der Praxis ist das nicht realistisch. Einige Dienste werden voneinander abhängig sein.

Es wird jedoch die Verwendung von Containern wie ‚Docker‘ genutzt, um diese Services in ihren eigenen Boxen zu kapseln. In jedem Container befindet sich die gesamte Anwendung mit all ihren Abhängigkeiten und Konfigurationsdateien, um sie auf anderen Computern ausführen zu können, ohne dass es zu Problemen aufgrund von Abhängigkeiten oder Versionsunterschieden kommen kann. Die Definition von ‚Docker‘ und ‚Containern‘ wird in diesem Kapitel noch genauer erläutert.

2.1.1 Microservices

Microservices bilden die Grundlage für eine Microservices-Architektur. Der Begriff veranschaulicht die Methoden zur Zerlegung einer Anwendung in allgemein kleine, in sich geschlossene Dienste, die in einer beliebigen Sprache geschrieben sind und über leichtgewichtige Protokolle kommunizieren. Mit unabhängigen Microservices können Softwareteams iterative Entwicklungsprozesse umsetzen sowie Funktionen flexibel erstellen und erweitern. Die Dienste sollten sorgfältig entkoppelt werden, um die Abhängigkeiten zwischen ihnen zu minimieren und die Autonomie der Dienste zu fördern. Außerdem sollten leichtgewichtige Kommunikationsmechanismen wie ‚REST‘ und ‚ttp‘ verwendet werden. Der ehemalige eBay- und Google-Manager Randy Shoup Jr. und Schmelmer 2016, S. 4–5 fasst die (drei) Merkmale von Microservices als drei Hauptmerkmale zusammen:

- Sie haben einen fokussierten Anwendungsbereich, in dem die Konzentration auf wenigen Funktionalitäten (manchmal nur einer einzigen) liegt, die durch eine kleine und ebenfalls gut definierte API ausgedrückt werden.
- Sie sind alle modular und unabhängig voneinander einsetzbar, wodurch die Regeln der Softwarekapselung und -modularität im Wesentlichen von der Programm- auf die Einsatzeinheitsebene angehoben werden.
- Schließlich verfügen Microservices über eine isolierte Persistenz, d. h., sie nutzen keinen gemeinsamen persistenten Speicher wie z. B. eine gemeinsame Datenbank.

Laut (Buelta 2019, S. 18) gibt es jedoch auch eine Reihe von Herausforderungen, die bei der Umstellung auf eine Microservice-Architektur auftreten können, von denen einige im Folgenden dargestellt werden:

1. Der Übergang wird wahrscheinlich schmerhaft sein, da ein pragmatischer Ansatz erforderlich ist und Kompromisse eingegangen werden müssen. Es wird auch eine Menge an Entwurfsdokumenten und Besprechungen erforderlich sein, um die Migration zu planen.
2. Es gibt auch eine Lernkurve beim Erlernen der Werkzeuge und Verfahren. Die Verwaltung von Clustern erfolgt anders als bei einem einzelnen Monolithen und die Entwickler müssen verstehen, wie die verschiedenen Services für lokale Tests zusammenarbeiten. Auch die Bereitstellung unterscheidet sich von der traditionellen, lokalen Entwicklung. Insbesondere das Erlernen von Docker erfordert einige Zeit der Anpassung.
3. Die Fehlersuche bei einer serviceübergreifenden Anfrage ist schwieriger als bei einem monolithischen System. Die Überwachung des Lebenszyklus einer Anfrage ist bedeutsam und einige subtile Fehler lassen sich in der Entwicklung nur schwer reproduzieren und beheben.
4. Die Aufteilung eines monolithischen Systems in verschiedene Services erfordert sorgfältige Überlegungen und eine gründliche Planung. Eine schlechte Trennlinie kann dazu führen, dass zwei Dienste eng miteinander gekoppelt sind.

5. Die Umstellung auf Microservices sollte mit Bedacht und unter sorgfältiger Abwägung der Vor- und Nachteile vorgenommen werden. Es ist möglich, dass es Jahre dauert, bis die Migration in einem ausgereiften System abgeschlossen ist.

In Anbetracht dieser Punkte ist es relevant, zu verstehen, wie bedeutsam es ist, die Aufgaben zu priorisieren und die Dinge langsam anzugehen, ohne sich darauf zu konzentrieren, die gesamte Anwendung auf einmal zu entwickeln, denn, wie bereits erwähnt, könnte dies viel Zeit in Anspruch nehmen.

2.1.2 Microservices-Implementierung mit Python

„Python ist eine erstaunlich vielseitige Sprache. Wie Sie wahrscheinlich bereits wissen, wird sie zum Erstellen vieler verschiedener Arten von Anwendungen verwendet - von einfachen Systemskripten, die Aufgaben auf einem Server ausführen, bis hin zu großen objektorientierten Anwendungen, die Dienste für Millionen von Benutzern ausführen. Laut einer Studie von Philip Guo aus dem Jahr 2014, die auf der Website der Association for Computing Machinery (ACM) veröffentlicht wurde, hat Python an den führenden US-Universitäten Java überholt und ist die beliebteste Sprache, um Informatik zu lernen. Dieser Trend setzt sich auch in der Softwarebranche fort. Im TIOBE-Index (<http://www.tiobe.com/tiobe-index/>) rangiert Python unter den fünf beliebtesten Sprachen, und in der Webentwicklung ist es wahrscheinlich noch beliebter, da Sprachen wie C nur selten als Hauptsprache für die Erstellung von Webanwendungen verwendet werden.“ (Ziadé 2017, S. 21)

2.2 Interkommunikation zwischen den Services

Normalerweise gibt es bei einer Microservice-Architektur einige Bestandteile, die erforderlich sind, damit die Services in der Lage sind, miteinander zu kommunizieren. Es gibt eine horizontale und eine vertikale Kommunikation. Die vertikale Kommunikation ist die Kommunikation, die zwischen den Services selbst stattfindet, und auf der anderen Seite haben wir die horizontale Kommunikation, die zwischen dem Client und der Anfrage stattfindet, die er an einen der gezielten Services innerhalb der Microservice-Architektur stellt.

2.2.1 Service-Discovery

Wenn eine vertikale Kommunikation stattfinden soll, müssen die Services in der Lage sein, herauszufinden, dass andere Services im Rahmen unserer Anwendung verfügbar sind. Normalerweise kann jeder Service die Verfügbarkeit eines anderen über einen Discovery-Service feststellen. Dieser Service registriert jeden Service innerhalb des Anwendungsbereichs und gibt ihm eine ID, damit andere Services ihn erkennen können. Es gibt einige Discovery-Services wie zum Beispiel ‚Consul‘, ‚Eureka‘ und ‚Kubernetes‘. In diesem Projekt wird jedoch ‚Docker Compose Networking‘ verwendet, um die Services füreinander auffindbar zu machen. Wenn hingegen eine horizontale Kommunikation angestrebt wird, sollte es eine Gateway-API geben, durch die die vom Client gestellte Anfrage an den richtigen Service weitergeleitet wird.

Dieses Gateway wird normalerweise konfiguriert und verfügt über einige Funktionen wie Sicherheit, Konnektivität, Skalierbarkeit sowie Verfügbarkeit. Die Entwicklung wird dadurch erleichtert. Das Gateway sollte dann eine Anfrage von einem Client entgegennehmen und sie an den richtigen Service weiterleiten, um die benötigte Ressource vom Server zurückzugeben.

Bisher hat sich gezeigt, dass für die Kommunikation zwischen Client, Server und Services mehr als eine Komponente erforderlich ist. Dies kann manchmal schwierig sein und die Komplexität der Entwicklung erhöhen. Das Ziel dieser Arbeit ist es, die Komplexität zu verringern, da dies der Hauptzweck der Microservices-Architektur ist. Dies erhöht jedoch die Komplexität bei der kurzfristigen Entwicklung und verringert sie auf lange Sicht, nachdem eine nachhaltige Microservice-Architektur aufgebaut worden ist. An dieser Stelle wurde nach Möglichkeiten gesucht, diese Kommunikation zu vereinfachen, ohne dabei zu riskieren, dass die erheblichen Funktionen, die diese anderen APIs zu bieten haben, verloren gehen. Während der Suche wurde ein Kreuzweg erreicht, bei dem zwischen ‚Service-Meshes‘, ‚API-Gateways‘ und ‚Advanced-Message-Queuing-Protocol‘ unterschieden werden musste, um zu einer optimalen Lösung zu gelangen. Wobei jede Technologie seinen eigenen Vorteil an einem bestimmten Punkt der Entwicklung hat.

Im Artikel von Wolfram 2019 wurden die Unterschiede zwischen ihnen erklärt und erläutert und danach wird zusammengefasst warum einige von ihnen für die Entwicklung des ‚Tennis Companion‘-Portals ausgewählt wurden.

2.2.2 Service Mesh

Laut der Artikel zufolge wurde erklärt, dass in einer Microservice-Architektur das Dienstnetz eine dynamische Nachrichtenschicht bildet, die die Kommunikation erleichtert. Service-Meshes sind dezentralisierte und selbstorganisierende Netzwerke zwischen Microservice-Instanzen, die den Lastausgleich, die Erkennung von Endpunkten, Zustandsprüfungen, die Überwachung und das Tracing übernehmen. Sie funktionieren, indem jeder Instanz ein kleiner Agent, ein sogenannter ‚Sidecar‘, zugeordnet wird, der den Datenverkehr vermittelt und die Registrierung der Instanzen, die Erfassung von Metriken und die Wartung übernimmt. Obwohl sie konzeptionell dezentralisiert sind, verfügen die meisten Dienstnetze über ein oder mehrere zentrale Elemente, die Daten sammeln oder Verwaltungsschnittstellen bereitstellen. Das bedeutet, dass die Entwickler bei der Erstellung der Anwendung keine prozessinterne Kommunikation programmieren müssen. Zwei der derzeit beliebtesten Service-Mesh-Optionen sind ‚Istio‘, ein Projekt, das Google zusammen mit IBM und Lyft gestartet hat, und ‚Linkerd‘, ein Projekt der Cloud Native Computing Foundation. Sowohl Istio als auch Linkerd sind mit Kubernetes verknüpft, unterscheiden sich jedoch in Bereichen wie der Unterstützung von Umgebungen ohne Container und den Funktionen zur Verkehrssteuerung. Die Vorteile von Service-Meshes liegen darin, dass sie dynamischer sind und sich leicht umgestalten und neue Funktionalitäten und Endpunkte aufnehmen lassen. Ihr dezentraler Charakter erleichtert die Arbeit an Microservices in relativ isolierten Teams. Sie haben jedoch auch Nachteile: Service-Meshes können recht komplex sein und erfordern eine Vielzahl beweglicher Teile. Die vollständige Nutzung von Istio erfordert beispielsweise die Bereitstellung eines separaten Traffic-Managers, eines

Telemetriesammlers, eines Zertifikatsmanagers und eines Sidecar-Prozesses für jeden Knoten. Es handelt sich außerdem um eine relativ neue Entwicklung, wodurch etwas, das das Rückgrat Ihrer Architektur bildet, beunruhigend jung ist.

2.2.3 Message Queues (MQ's)

Im gleichen Artikel von Wolfram 2019 wurde den Unterschied zwischen Nachrichten-Warteschlangen und Mesh-Services als zwei völlig unterschiedliche Dinge beschrieben; sie lösen jedoch dasselbe Problem auf unterschiedliche Weise. Nachrichten-Warteschlangen ermöglichen die Einrichtung komplexer Kommunikationsmuster zwischen Diensten durch Entkopplung von Sender und Empfänger. Sie erreichen dies durch eine Reihe von Konzepten, wie z. B. themenbasiertes Routing oder Publish-Subscribe-Messaging sowie gepufferte Task-Warteschlangen, die es mehreren Instanzen leicht machen, verschiedene Aspekte einer Aufgabe im Laufe der Zeit zu verarbeiten. Zu den Vorteilen von MQ's gehört, dass es durch das Konzept der Entkopplung von Sender und Empfänger eine Reihe anderer Konzepte wie Zustandsprüfungen, Routing, Endpunktsuche oder Lastausgleich überflüssig machen kann. Instanzen können relevante Aufgaben aus einer gepufferten Warteschlange abrufen, wenn sie dazu bereit sind. Dies wird besonders leistungsfähig, wenn die automatische Orchestrierung und Skalierungsentscheidungen auf der Anzahl der Nachrichten in jeder Warteschlange basieren, was zu äußerst ressourceneffizienten Systemen führt. Einige der beliebtesten offenen Quellen für MQs sind Apache Kafka, AMQP-Broker wie RabbitMQ oder HornetQ und Cloud-Provider-Versionen wie AWS SQS oder Kinesis, Google PubSub oder Azure Service Bus. Im folgenden Unterabschnitt wird ein von Akshay Kamath B und Chaitra B H veröffentlichtes Papier erörtert, in dem dargelegt wird, warum der AMQP-Broker RabbitMQ einen klaren Vorteil gegenüber der REST-API für die Kommunikation zwischen Micro-Services-Webanwendungen hat.

2.2.4 API Gateways

Laut ebd. ist ein API-Gateway der ‚größere Bruder‘ des alten Reverse-Proxys für HTTP-Aufrufe. Es ist ein skalierbarer, in der Regel weborientierter Server, der Anfragen sowohl aus dem öffentlichen Internet als auch von internen Diensten empfangen und an die am besten geeignete Microservice-Instanz weiterleiten kann. API-Gateways verfügen in der Regel über eine Reihe hilfreicher Funktionen, darunter Lastausgleich und Zustandsprüfungen, API-Versionierung und -Routing, Authentifizierung und Autorisierung von Anfragen, Datentransformation, Analysen, Protokollierung, SSL-Terminierung und mehr. Beispiele für beliebte Open-Source-API-Gateways sind ‚Kong‘ oder ‚Tyk‘. In diesem Projekt wird ‚NGINX‘ als API-Gateway verwendet, denn es hat den Vorteil, dass es diese Rolle eines Standard-API-Gateways übernehmen kann und gleichzeitig als Reverse-Proxy, Load-Balancer sowie Webserver für den bestehenden HTTP-Verkehr fungiert. (ebd.)

In diesem Projekt wird NGINX als API-Gateway verwendet, denn es hat den Vorteil, dass es diese Rolle eines Standard-API-Gateways übernehmen kann und gleichzeitig als Reverse Proxy, Load Balancer und Webserver für den bestehenden HTTP-Verkehr fungiert.

2.2.5 RabbitMQ

RabbitMQ ist leichtgewichtig und kann sowohl vor Ort als auch in der Cloud eingesetzt werden. Es unterstützt mehrere Messaging-Protokolle. RabbitMQ kann in verteilten und föderierten Konfigurationen eingesetzt werden, um hohe Anforderungen an die Verfügbarkeit zu erfüllen. Es läuft auf vielen Betriebssystemen und Cloud-Umgebungen und bietet eine breite Palette von Entwickler-Tools für die meisten gängigen Sprachen. (RabbitMQDocu n.d.[a])

Ursprünglich wurde das Advance-Message-Queuing-Protocol implementiert und seitdem mit einer Plug-in-Architektur erweitert, um das Streaming-Text-Oriented-Messaging Protocol (STOMP), MQ-Telemetry-Transport (MQTT) und andere Protokolle zu unterstützen. (RabbitMQDocu n.d.[b])

Unterschiede zwischen RESTful und RabbitMQ

Hier werden die Eigenschaften von RabbitMQ ausführlicher besprochen. RabbitMQ bietet einige Features, die es für eine asynchrone Kommunikation kompatibel machen, bei der ein Publisher eine Nachricht an RabbitMQ veröffentlichen kann, ohne sich Gedanken darüber zu machen, ob ein anderer Dienst bereit ist, sie zu empfangen. Einige dieser Funktionen, wie sie in der Abhandlung von Akshay Kamath B und Chaitra B H B und H 2020 erklärt werden, sind:

1. Verlässlichkeit: RabbitMQ wird separat gehostet und läuft unabhängig von dem Micro-Service, der es verwendet. Es verfügt über eine dauerhafte Bereitstellung und bietet eine zuverlässige Funktionalität.
2. Flexible Weiterleitung: RabbitMQ unterstützt das Routing durch Exchanges. Es gibt viele Arten von Exchanges in RabbitMQ, die verwendet werden kann, um das gewünschte Routing-System aufzubauen. Es unterstützt auch das Schreiben eigener Exchanges durch Plugins.
3. Hohe Verfügbarkeit: Die Warteschlange wird auf mehreren Rechnern in einem Cluster gespiegelt. Jeder Ausfall der Hardware stellt sicher, dass es ein Backup der Daten gibt und die Nachrichten sicher sind.
4. Plattformunabhängig: Wie bereits erwähnt wird RabbitMQ separat gehostet und der Micro-Service, der es benutzt, kann in jeder Sprache gehalten sein. In den meisten Fällen werden Publisher- und Subscriber-Micro-Service in einer anderen Sprache geschrieben und implementiert.

Mit Blick auf die oben genannten Eigenschaften stellt sich nun die bedeutsame Frage, welche Vorteile RabbitMQ gegenüber einem Representational-State-Transfer (REST) hat. In der Abhandlung werden sechs Vorteile genannt, warum ein Entwickler mit RabbitMQ arbeiten sollte, um eine optimale Kommunikation zwischen dem Client und den Diensten innerhalb der zu erstellenden Webanwendung zu ermöglichen.

Die sechs Vorteile sind in der gleichen wissenschaftlichen Zeitschrift wie folgt beschrieben:

1. Message Queue bietet ein asynchrones Kommunikationsprotokoll. Published kann eine Nachricht in RabbitMQ veröffentlichen, ohne sich darum zu kümmern, ob ein anderer Dienst bereit ist, sie zu empfangen.
2. Die Nachricht wird in der Warteschlange gespeichert, bis der Dienst nicht Anbieter ist, falls der Verbraucher langsamer ist als der Produzent wartet die Nachricht in der Warteschlange.
3. Die Nachricht, die den Dienst veröffentlicht, muss nicht wissen, wie der Dienst arbeitet, der sie verarbeiten wird.
4. RabbitMQ hält den Producer und den Consumer unabhängig voneinander.
5. RabbitMQ hat eine eingebaute Benutzeroberfläche, die es erlaubt, alle in der Warteschlange stehenden Nachrichten und laufenden Prozesse zu sehen. Der Datenverkehr, der in die Warteschlange eingeht, kann live verfolgt werden.

2.3 Agile Entwicklung mit Scrums

Da das Projekt nur aus einer Person besteht, wird die agile Entwicklung nicht im Detail besprochen, aber es wird angestrebt, das Projekt mit agilen Ansätzen abzuschließen. Lediglich die Definition der Anforderungen wird kurz geklärt, da sie in einer späteren Phase des Projekts verwendet werden soll.

Anforderungen

„Anforderungen werden in Scrum im zentralen Product Backlog verwaltet. Das Product Backlog enthält eine List mit priorisierten und geschätzten User Stories. Eine User Story ist eine in der Sprache des Kunden beschriebene Anforderung an das System, die einen konkreten und für den Kunden sichtbaren Mehrwert liefert. Für das Schreiben der User Stories und deren Verwaltung im Product Backlog ist der Product Owner zuständig. Nur er darf neue Stories hinzufügen, die Priorität vorhandener Stories ändern oder auch mit der Zeit unwichtig gewordene Stories aus dem Backlog entfernen. Natürlich kann der Product Owner das Product Backlog nicht ganz alleine füllen. Dafür gibt es zu viele wichtige Ideen anderer Interessenvertreter, die der Product Owner zu Beginn des Projektes einen Anforderungsworkshop mit allen an „Tennis Companion“ Portal interessierten Personen durch. Gemeinsam wird eine Mindmap mit den zentralen Ideen und Inhalten der Plattform erstellt. Dazu schlüpft das Anforderungsteam in die verschiedenen Benutzerrollen (graue Kätschen in Abbildung 2.3.1) und überlegt sich, was Scrum-Coaches und Projektanbieter, aber auch ehemalige Auftraggeber oder Administratoren mit dem System erreichen wollen.“ (Wirdemann und Mainusch 2017, S. 10)

Im Anschluss an den Workshop zieht sich der Product-Owner einige Stunden zurück und formuliert auf Basis der erstellten Mindmap eine Reihe von User-Storys und trägt sie ins

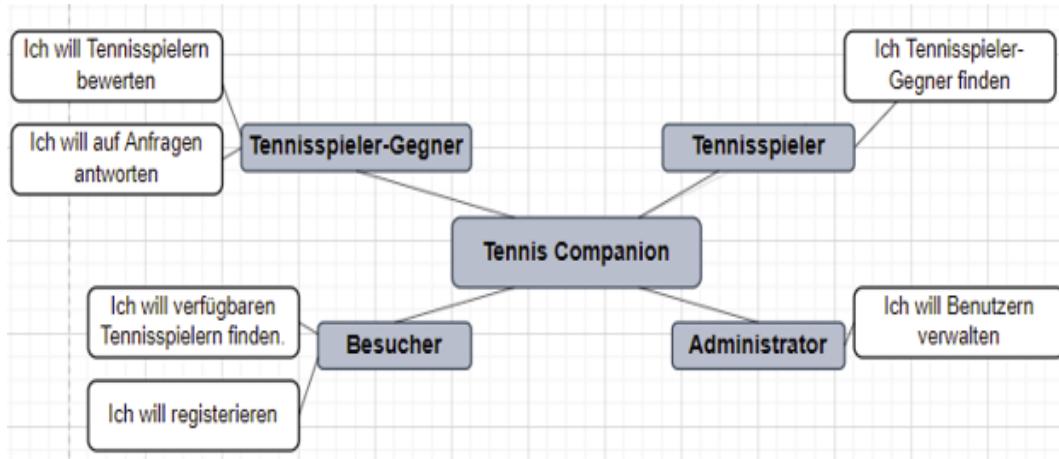


Abbildung 2.3.1: User Stories und Epics (Quelle: eigene Darstellung)

Product-Backlog ein. Die Reihe User-Storys werden ausführlicher im nächsten Kapitel diskutiert und durchgelistet.

2.4 Frameworks im Projektbereich

2.4.1 Django

„Django ist ein Python-Web-Framework, das sowohl ein Open-Source-Framework auf hohem Niveau bietet als auch „schnelle Entwicklung und sauberes, pragmatisches Design fördert“. Es ist schnell, sicher und skalierbar. Django bietet einen starken Community-Support und eine ausführliche Dokumentation. Darüber hinaus ist es flexibel, so dass mit MVPs bis hin zu größeren Unternehmen gearbeitet werden kann. Einige der größten Unternehmen, die Django verwenden, sind Instagram, Dropbox, Pinterest und Spotify. Ein Django-Projekt besteht in der Regel aus vielen Anwendungen, die in `INSTALLED_APPS` deklariert sind. Django-Anwendungen sollten der Unix-Philosophie „Do one thing and do it well“ folgen, wobei der Schwerpunkt darauf liegt, klein und modular zu sein, was die Django-Designphilosophie der ‚losen Kopplung‘ widerspiegelt.“ (Djangodocu 2021)

Settings

Das Settings-Modul ist die einzige echte Voraussetzung für ein Django-Projekt. Normalerweise befindet es sich in der Wurzel des Projekts als `settings.py`.

- **Umgang mit Dateipfaden** (en: handling file paths):

Eine Funktion der Einstellungen ermöglicht es, Django mitzuteilen, wo es z. B. alle statischen Medien und Vorlagen finden soll. Höchstwahrscheinlich befinden sie sich bereits innerhalb des Projekts. Wenn ja, lässt sich die absoluten Pfadnamen von Python generieren. Dies macht das Projekt über verschiedene Umgebungen (en: environments) hinweg portabel.

```
import os
DIRNAME = os.path.dirname(__file__)
#...
STATIC_ROOT = os.path.join(DIRNAME, 'static')
```

Django MVT

Das Model View Template (MVT) ist ein Software-Entwurfsmuster. Es ist eine Sammlung von drei wesentlichen Komponenten Model, View und Template. Das Model hilft bei der Handhabung der Datenbank. Es ist eine Datenzugriffsschicht, die die Daten verarbeitet. Die Schablone ist eine Präsentationsschicht, die den Teil der Benutzeroberfläche vollständig übernimmt. Die View-Komponente wird verwendet, um die Geschäftslogik auszuführen und mit einem Modell zu interagieren, um Daten zu übertragen und eine Vorlage zu rendern. Obwohl Django dem Model-View-Controller(MVC)-Muster folgt, behält es seine eigenen Konventionen bei. So wird die Steuerung durch das Framework selbst übernommen. Es gibt keinen separaten Controller und die gesamte Anwendung basiert auf Model, View und

Template. Deshalb wird sie auch MVT-Anwendung genannt. Das folgende Diagramm zeigt den MVT-basierten Kontrollfluss:

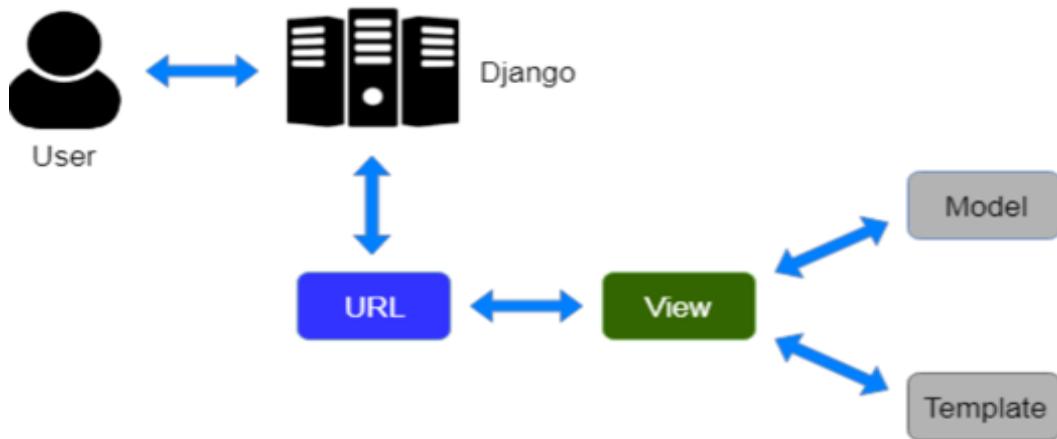


Abbildung 2.4.1: Skizze einer DJANGO-MVT-Struktur, die erstellt wird aus einem Javapoint-Artikel (Jayanandana 2018).

Hier fordert ein Benutzer eine Ressource bei Django an. Django arbeitet als Controller und prüft die verfügbare Ressource in der URL. Wenn URL-Maps, eine Ansicht aufgerufen wird, die mit Modell und Vorlage interagieren, rendert es eine Vorlage. Django antwortet dem Benutzer zurück und sendet eine Vorlage als Antwort. (javatpoint 2022) In diesem Projekt wird ReactJS als Frontend fungieren und Within wird die Vorlage für den Endbenutzer darstellen, um mit der Anwendung zu interagieren.

2.4.2 ReactJS

React, oft auch als React.js oder ReactJS bezeichnet, ist eine deklarative JavaScript-Bibliothek für die Erstellung leistungsstarker Benutzeroberflächen im Frontend. Es handelt sich um ein Open-Source-Projekt, das 2011 von Jordan Walke und seinem Team bei Facebook entwickelt wurde. Mit React können wir komplexe Benutzeroberflächen aus wiederverwendbaren, benutzerdefinierten Komponenten erstellen, die kleine und isolierte Code-Blöcke sind. Das macht sie effizient und flexibel. Außerdem vereinfacht React die Speicherung und Verarbeitung von Daten mithilfe von Zuständen und Requisiten. Es wird oft als Framework missverstanden, aber es ist kein Framework. Es ist die Ansichtsschicht einer MVC-Anwendung im Gegensatz zu AngularJS, das ein vollständiges Web-Framework ist.

Warum ReactJS

Laut der Stack Overflow 2020 Developer Survey (Overflow 2020) wird React zu einer der beliebtesten Webtechnologien und lässt Angular, ASP.NET, Express, Vue.js, Django, Spring und Flask hinter sich. Auch in der Liste der meistgewünschten Webtechnologien rangiert sie ganz oben. Sie verfügt über eine reiche Gemeinschaft von Millionen von Entwicklern, mit

mehr als 1389 Mitwirkenden und 150 000 Stars auf GitHub. Im Folgenden ist ein Google-Trends-Vergleich abgebildet, der zeigt, wie beliebt React in den letzten fünf Jahren gegenüber zu anderen Web-Frameworks geworden ist.

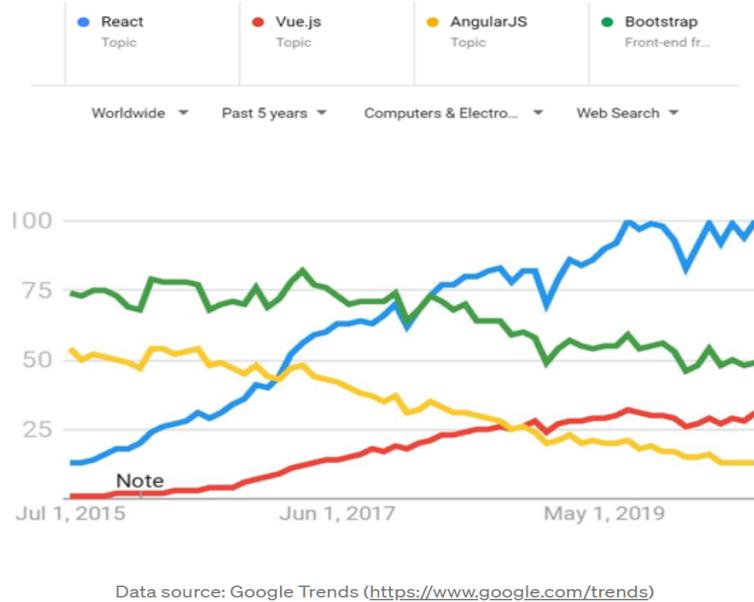


Abbildung 2.4.2: ReactJS Bekanntheitsgrad von 2015 bis 2020 aus (Ram 2021)

Es wäre keine Übertreibung zu sagen, dass React die Zukunft der Front-End-Entwicklung sein wird, da Websites von erfolgreichen Unternehmen wie Netflix, Airbnb, Reddit, Tesla, Dropbox, BBC, Facebook, Instagram, Khan Academy, Uber, PayPal, Twitter, Yahoo Mail und OkCupid Front-Ends für ihre Apps verwenden und diese damit betrieben werden. (Ram 2021)

ReactJS Vorteile

Merkmale von React:

- Es ist komponentenbasiert und bietet Wiederverwendbarkeit und fördert freundliche Kodierung.
- Es basiert auf Sprachen wie HTML und JSK und ist daher einfach zu verwenden.
- Es verwendet ein virtuelles DOM, das nur Teile der Webseite erneuert und so die Neuaufbauzeit reduziert.
- Es handelt sich um eine deklarative Programmierung, die es dem Entwickler erleichtert, die Ausgabe des Browsers vorherzusagen im Gegensatz zu Letzterem. Daher ist es einfach zu verstehen, wie die Ansicht gerendert wurde, was die Geschwindigkeit der Fehlersuche erhöht.

- Es ist gut, eine dynamische und reaktionsfähige Benutzeroberfläche für jede Webschnittstelle mit hoher Leistung zu erstellen. Laut Forbes kann eine großartige Benutzeroberfläche die Konversionsrate Ihrer Website um 200 % steigern.
- Wenn Sie in Zukunft React Native für die Erstellung reaktionsfähiger, agiler, nativer mobiler Anwendungen verwenden möchten, wäre es für Sie vergleichsweise einfacher, es zu lernen, da es auf den Prinzipien von React basiert.(Ram 2021)

Create React App

Im gleichen Artikel ebd. wurde auch ‚Create React App‘ als eine Software von Facebook erläutert, die alles enthält, was man zum Erstellen einer React-App braucht. Es funktioniert auf Windows, macOS und Linux. Es verwendet Babel und webpack. Sie sollten bedenken, dass es keine Datenbankverwaltung oder den Back-End-Teil übernimmt. Es erlaubt Ihnen nur, ein Frontend zu erstellen, das mit einem Backend Ihrer Wahl kombiniert werden kann. Es erstellt einen Live-Entwicklungsserver und kompiliert automatisch React, setzt automatisch CSS-Dateien, JSX (JavaScript XML) und ES6 (ECMAScript 6) als Vorzeichen und testet den Code und warnt vor Fehlern mit ESLint.

In dem gleichen Medium-Artikel ebd. wird es beschrieben wie einfach es ist, eine React-App zu erstellen. Es wird lediglich ein npm-Paketmanager benötigt, über den das npx-Paket verwendet werden kann, ein Paketausführungswerkzeug, das mit npm geliefert wird, um eine React-App mit einem freigewählten Namen zu erstellen:

Es ist sehr einfach, eine React-App zu erstellen, es wird lediglich ein npm-Paketmanager benötigt, über den man das npx-Paket verwenden kann, ein Paketausführungswerkzeug, das mit npm geliefert wird, um eine React-App mit dem Namen Ihrer Wahl zu erstellen, Beispiel:

```
npx create-react-app app_name
```

2.5 Containers

Wie in einem Online-Artikel (Clany 2021) über Container erwähnt, sind Container Softwareeinheiten, die Dienste und ihre Abhängigkeiten bündeln und eine konsistente Einheit für Entwicklung, Test und Produktion bilden. Container sind für die Bereitstellung von Microservices nicht erforderlich und auch Microservices sind für die Verwendung von Containern nicht notwendig. Container können jedoch die Bereitstellungszeit und die Anwendungseffizienz in einer Microservices-Architektur potenziell verbessern, und zwar stärker als andere Bereitstellungstechniken, z. B. VMs. Der Hauptunterschied zwischen Containern und VMs besteht darin, dass Container ein Betriebssystem und Middleware-Komponenten gemeinsam nutzen können, während jede VM ein komplettes Betriebssystem für ihre Verwendung enthält. Da nicht jede VM ein eigenes Betriebssystem für jeden kleinen Dienst bereitstellen muss, können Unternehmen eine größere Sammlung von Microservices auf einem einzigen Server ausführen. Ein weiterer Vorteil von Containern besteht darin, dass sie nach Bedarf eingesetzt werden

können, ohne die Anwendungsleistung zu beeinträchtigen. Entwickler können sie außerdem mit relativ geringem Aufwand ersetzen, verschieben und replizieren. Die Unabhängigkeit und Konsistenz von Containern ist ein entscheidender Faktor für die Skalierung bestimmter Teile einer Microservices-Architektur – je nach Arbeitslast – und nicht der gesamten Anwendung. Sie unterstützt auch die Fähigkeit, Microservices bei einem Ausfall neu zu verteilen. Aus diesem Grund sind Container leichtgewichtig und im Gegensatz zu virtuellen Maschinen, bei denen es bis zu Minuten dauern kann, eine Anwendung auszuführen, dauert es nur Sekunden, um einen Container auszuführen. Eine Illustration der Infrastruktur eines Containers ist in der folgenden Abbildung zu sehen:

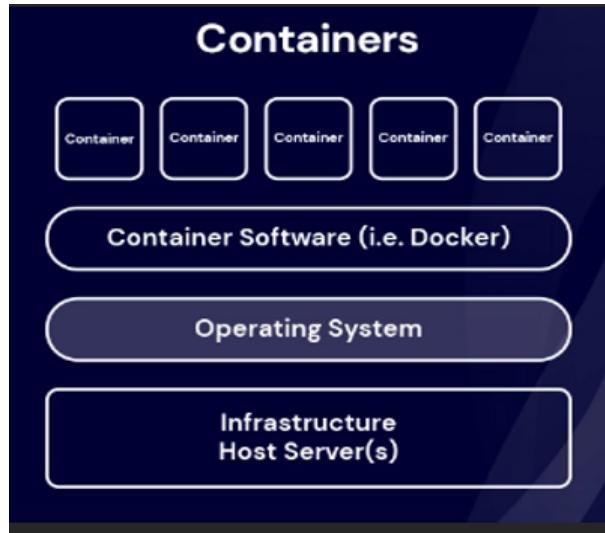


Abbildung 2.5.1: Skizze der Infrastruktur eines Containers aus einem Backblaze-Artikel (Clany 2021)

Im Vergleich zu älteren Virtualisierungstechnologien, nämlich virtuellen Maschinen, die in Gigabytes gemessen werden, sind Container nur Megabytes groß. Das bedeutet, dass eine ganze Reihe von ihnen auf einem bestimmten Computer oder Server ausgeführt werden kann, so wie viele Container auf einem Schiff gestapelt werden können. Container, vor allem über Docker Popularität, hat das Problem der Umgebung Inkonsistenz für Entwickler gelöst. Docker, das als Open-Source-Plattform für das Container-Management begann, ist einer der bekanntesten Anbieter im Container-Bereich. Der Erfolg von Docker hat jedoch dazu geführt, dass sich ein großes Tooling-Ökosystem um Docker herum entwickelt hat, aus dem beliebte Container-Orchestratoren wie Kubernetes hervorgegangen sind. Docker wird auch die wesentliche Software sein, die in dieser Arbeit verwendet wird, um Dienste unabhängig voneinander auszuführen. So kann eine ganze Anwendung mit ihren eigenen Abhängigkeiten und Konfigurationen in einem eigenen Container ausgeführt werden, was das Problem der Inkonsistenz der Umgebung für die Entwickler löst und die Skalierbarkeit der gesamten Anwendung erhöht, indem weitere Container hinzugefügt werden können, wenn eine neue Funktion benötigt wird. Diese neue Funktion kann dann von einem anderen Team oder einer anderen Person in einem eigenen unabhängigen Container ergänzt werden und,

sobald sie fertiggestellt ist, kann sie eingefügt werden, um mit den übrigen Containern über unsere Microservice-Architektur zu kommunizieren. „Docker und Kubernetes sind nicht die einzigen Optionen da draußen, aber in ihrer Umfrage aus dem Jahr 2021 fand Stack Overflow heraus, dass fast 50 % der über 76.000 Befragten Docker und 17 % Kubernetes verwenden. Aber was tun sie?“ (Clany 2021)

2.5.1 Docker

Die Containertechnologie gibt es schon seit einiger Zeit in Form von Linux-Containern oder LXC, aber eine Verbreitung von Containern erfolgte erst in den letzten fünf bis sieben Jahren mit der Einführung von Docker. Docker wurde 2013 als Projekt zur Entwicklung von LXC-Containern für eine einzige Anwendung ins Leben gerufen und führte zu mehreren Änderungen an LXC, die Container portabler und flexibler machen. Später entwickelte es sich zu einer eigenen Container-Laufzeitumgebung. Auf einer hohen Ebene ist Docker ein Linux-Dienstprogramm, mit dem Container effizient erstellt, ausgeliefert und ausgeführt werden können. (ebd.)

„Docker ist eine Reihe von Platform-as-a-Service(PaaS)-Produkten, die Virtualisierung auf Betriebssystemebene nutzen, um Software in Paketen, so genannten Containern, bereitzustellen“ (Maureen 2013). Laut Docker(FAQ) 2019 sind Container voneinander isoliert und bündeln ihre eigene Software, Bibliotheken und Konfigurationsdateien; sie können über genau definierte Kanäle miteinander kommunizieren. Da alle Container die Dienste eines einzigen Betriebssystemkerns nutzen, verbrauchen sie weniger Ressourcen als virtuelle Maschinen.(Docker 2022)

„Docker macht die Entwicklung effizient und vorhersehbar; es nimmt sich wiederholende, banale Konfigurationsaufgaben weg und wird während des gesamten Entwicklungslebenszyklus für eine schnelle, einfache und portable Anwendungsentwicklung verwendet – auf dem Desktop und in der Cloud. Die umfassende End-to-End-Plattform von Docker umfasst Benutzeroberflächen, Befehlszeilen (CLIs), APIs und Sicherheitsfunktionen, die so konzipiert sind, dass sie über den gesamten Lebenszyklus der Anwendungsentwicklung zusammenarbeiten.“ Wie bereits erwähnt, kann Docker eine Anwendung und ihre Abhängigkeiten in einen virtuellen Container packen, der auf jedem Linux-, Windows- oder macOS-Computer einsetzbar ist. „Dadurch kann die Anwendung an verschiedenen Orten ausgeführt werden, z. B. vor Ort, in einer öffentlichen oder privaten Cloud.“ (DockerDocu 2022a) Wie bereits erwähnt, kann Docker eine Anwendung und ihre Abhängigkeiten in einen virtuellen Container packen, der auf jedem Linux-, Windows- oder macOS-Computer ausgeführt werden kann. „Dadurch kann die Anwendung an verschiedenen Orten ausgeführt werden, z. B. vor Ort, in einer öffentlichen oder privaten Cloud.“(Noyes 2013)

Nach den Worten von Buelta 2019, dem Autor des Buches „Hands-On-Docker“, ist die Microservice-Architektur relativ unabhängig von der Plattform, die sie unterstützt. Sie kann auf alten physischen Boxen in einem dedizierten Rechenzentrum, in einer öffentlichen Cloud oder in containerisierter Form bereitgestellt werden. Es besteht jedoch die Tendenz, Contai-

ner für die Bereitstellung von Microservices zu verwenden. Bei Containern handelt es sich um ein paketiertes Softwarebündel, das alles, was zum Ausführen erforderlich ist, einschließlich aller Abhängigkeiten, kapselt. Sie benötigen lediglich einen kompatiblen Betriebssystem-Kernel, um autonom zu laufen. Die Arbeit mit Docker-Containern besteht aus zwei Schritten. Zunächst wird der Container erstellt, indem Schicht für Schicht von Änderungen im Dateisystem vorgenommen werden, z. B. das Hinzufügen der auszuführenden Software und Konfigurationsdateien. Dann führen wir ihn aus, indem wir seinen Hauptbefehl starten. Die Microservices-Architektur passt gut zu einigen der Merkmale von Docker-Containern: kleine Einzweck-Elemente, die über HTTP-Aufrufe kommunizieren. Deshalb werden sie heutzutage in der Regel gemeinsam präsentiert, auch wenn dies nicht zwingend erforderlich ist. Ein wesentlicher Faktor für den Umgang mit Containern ist, dass sie zustandslos sein sollten. Jeder Zustand muss in einer Datenbank gespeichert werden und kein Container speichert dauerhafte Daten. Dies ist eines der Schlüsselemente für skalierbare Webserver, was bei einer Anzahl von Servern nicht unbedingt der Fall ist. Ein weiterer Vorteil von Docker ist die Verfügbarkeit einer Vielzahl gebrauchsfertiger Container. Docker Hub ist ein öffentliches Register voller interessanter Container, die geerbt oder direkt verwendet werden können, entweder in der Entwicklung oder in der Produktion.

Die Ausführung von Docker in unserer Anwendung erfordert ein Verständnis zweier relevanter Dateien, die hinzugefügt und konfiguriert werden müssen, damit die Dienste auf Docker in einem eigenen Container mit dem entsprechenden Image laufen und schließlich auch mit anderen Containern kommunizieren können. Jeder Container hat eine einzige Konfigurationsdatei, das Dockerfile, in der sich alle Anweisungen zur Erstellung des Images befinden.

Zu Beginn muss ein Dockerfile geschrieben werden, um darzustellen, was in das Image aufgenommen werden soll: ein Python-Projekt, ein Java-Projekt, eine PostgreSQL-Datenbank usw. Dann kann aus dieser Dockerfile ein Image erstellt und von diesem können ein oder mehrere Container gestartet werden. Was ist der Unterschied zwischen Images und Containern? Wie in der obigen Abbildung dargestellt, könnte es wie in der OOP gesehen werden: Das Image ist die Klasse und der Container ist das Objekt. Es können mehrere Objekte aus einer einzigen Klasse instanziert werden, wobei jedes von ihnen seine eigenen Daten besitzt. Diese Metapher zugrunde gelegt, kann gesagt werden, dass das Dockerfile dem Code entspricht, der zur Erläuterung einer Klasse geschrieben wird. Im Folgenden werden noch einige Details zum Dockerfile und zur docker-composer.yml besprochen.

Dockerfile

In der Docker-Dokumentation wird ein DockerFile wie folgt erklärt:

„Docker kann Images automatisch erstellen, indem es die Anweisungen aus einem Dockerfile liest. Dies ist ein Textdokument, das alle Befehle enthält, die ein Benutzer auf der Kommandozeile aufrufen kann, um ein Image zusammenzustellen. Mit Docker-Build können Benutzer einen automatischen Build erstellen, der mehrere Befehlszeilenanweisungen nacheinander ausführt.“ (DockerDocu 2022b)

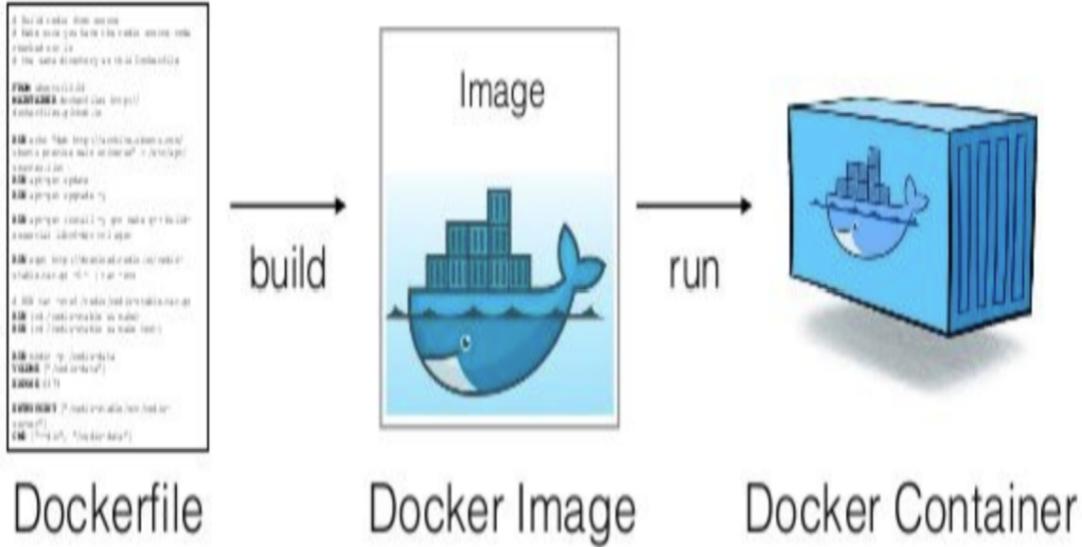


Abbildung 2.5.2: Skizze eines anhand eines Mediumartikels erstellten Docker-Images. Es wird in eine Datei namens Dockerfile geschrieben und in ein Docker-Image übersetzt, bevor aus dem Image ein Container erstellt wird. (Jayanandana 2018)

„Es beschreibt den Schritt, welche zur Erzeugung eines Docker-Image führen. Der Informationsfluss folgt dabei dem zentralen Schema: Dockerfile > Docker-Image > Docker-Container.“ (KnowHow 2021)

In einem Artikel von Ionos (ebd.) wird der Aufbau eines Dockerfiles wie folgt beschrieben:

Ein Dockerfile definiert die Schritte zur Erzeugung eines neuen Images. Wichtig zu verstehen ist dabei, dass immer mit einem existierenden Basis-Image (FROM <Parent Image>) begonnen wird. Das neu erzeugte Image erbt vom Basis-Image. Hinzu kommt eine Reihe punktueller Änderungen. Ein Dockerfile legt für ein neues Docker-Image zwei Punkte fest:

1. das Basis-Image, von dem das neue Image abstammt. Hiermit wird das neue Image im Stammbaum des Docker-Ökosystems verankert.
2. eine Reihe spezifischer Änderungen, die das neue Image vom Basis-Image unterscheiden.

Ein Dockerfile stellt eine Plain-Text-Datei mit dem Dateinamen „Dockerfile“ dar. Die Großschreibung des ersten Buchstabens ist vorgeschrieben. Diese Datei enthält einen Eintrag pro Zeile. Kommentare in einem Dockerfile beginnen mit einem Rautezeichen. Damit Kommentare Anerkennung finden, müssen sie an den Anfang der Zeile geschrieben werden. In einem Dockerfile gibt es verschiedene Anweisungen. Es muss mit der Anweisung FROM beginnen. In der untenstehenden Abbildung sind einige bedeutsame Anweisungen aufgeführt, die zu einem Dockerfile gehören:

Anweisung	Beschreibung	Kommentar
FROM	Basis-Image festlegen	Muss als erste Anweisung auftreten; nur ein Eintrag pro Build-Stage
ENV	Umgebungsvariablen für Build-Prozess und Container-Laufzeit setzen	–
ARG	Kommandozeilen-Parameter für Build-Prozess deklarieren	Darf vor FROM-Anweisung auftreten
WORKDIR	Aktuelles Verzeichnis wechseln	–
USER	Nutzer und Gruppenzugehörigkeit wechseln	–
COPY	Dateien und Verzeichnisse in das Image kopieren	Legt neuen Layer an
ADD	Dateien und Verzeichnisse in das Image kopieren	Legt neuen Layer an; von Nutzung wird abgeraten
RUN	Befehl im Image während des Build-Prozesses ausführen	Legt neuen Layer an
CMD	Standard-Argumente für Container-Start festlegen	Nur ein Eintrag pro Build-Stage
ENTRYPOINT	Standard-Befehl für Container-Start festlegen	Nur ein Eintrag pro Build-Stage
EXPOSE	Port-Zuweisungen für laufenden Container definieren	Ports müssen beim Starten des Containers aktiv geschaltet werden
VOLUME	Verzeichnis im Image beim Start des Containers im Host-System als Volumen einbinden	–

Abbildung 2.5.3: Befehle im Dockerfile nach dem Artikel von Ionos (KnowHow 2021)

Die FROM-Anweisung legt das Basis-Image fest, auf dem die nachfolgenden Anweisungen operieren. Sie darf pro Build-Stage nur einmal vorhanden sein und muss als erste Anweisung auftreten. Dies stellt die einzige Einschränkung dar. Die ARG-Anweisung kann vor der FORM-Anweisung auftreten. Jedem Docker-Image muss ein Basis-Image zugrunde liegen, anders ausgedrückt: Jedes Docker-Image hat genau ein Vorgänger-Image.

Docker Composer

Der Docker-Composer ist ein hilfreiches Werkzeug. So lassen sich zum Beispiel die zu erstellenden Images und die zu instanzierenden Container angeben, die als Ports gemappt werden sollen. Es handelt sich um eine einzige Konfigurationsdatei für das Projekt. Der Composer erstellt auch ein Docker-Netzwerk, damit die Container über ihre Namen miteinander kommunizieren können. Ein Beispiel sieht wie folgt aus:

```
'version: '3'  
services:  
  web:  
    build: .  
    ports: \5000:5000"  
  redis:  
    image: \redis:alpine'"
```

Die Autorin Buelta 2019, S. 28 erklärt, um mit einem Container in einer Clusteroperation zu arbeiten, wird im Allgemeinen „docker-compose“ verwendet. Dies ist das Docker-engine Orchestrierungswerkzeug zur Definition von Multi-Container-Operationen. Es wird durch eine YAML-Datei mit all den verschiedenen Aufgaben und Diensten definiert, jeder mit genügend Kontext, um ihn zu erstellen und auszuführen. Wie man einen Dienst in der Datei docker-compose.yaml definiert, wird im Kapitel über die Implementierung näher erläutert.

Docker innerhalb des Projekts

In diesem Projekt wird jeder Microservice in einer Docker-Compose-Datei gekapselt. Diese enthält alle Container (wie Datenbank, Nginx und RabbitMQ), die notwendigen sind, um den Microservice ausführen zu können.

2.5.2 Kubernetes

Als die Containerisierung ihren Höhepunkt erreichte, sahen sich viele frühe Anwender mit einem neuen Problem konfrontiert: Wie lässt sich eine ganze Reihe von Containern verwalten? Kubernetes ist ein Open-Source-Container-Orchestrator. Er wurde von Google (die Bereitstellung von Millionen von Containern pro Woche ist keine kleine Aufgabe) als „Minimum Viable Product“-Version des ursprünglichen Cluster-Orchestrators mit dem Namen Borg entwickelt. Kubernetes ermöglicht es Entwicklern, den gewünschten Zustand eines Container-Deployments mithilfe von YAML-Dateien zu beschreiben (YAML steht für Yet Another

Markup Language). Die YAML-Datei verwendet eine deklarative Sprache, um Kubernetes mitzuteilen, wie dieses Container-Deployment aussehen soll, und Kubernetes übernimmt die gesamte Arbeit des Erstellens und Verwaltens dieses Zustands.

2.6 Travis-CI

Die Autorin Buelta 2019, S. 115 beschreibt Travis-CI, Travis-CI (<https://travis-ci.com/>) ist ein beliebter Service, der zur kontinuierlichen Integration dient und für öffentliche GitHub-Projekte frei verfügbar ist. Die Integration in GitHub lässt sich leicht umsetzen. Sie ermöglicht es, die Plattform zu konfigurieren, auf der er läuft, beispielsweise macOS, Linux oder iOS. Travis-CI ist eng mit GitHub verknüpft, sodass eine Anmeldung bei GitHub genügt, um darauf zugreifen zu können.

Die Links zur konfigurierten Travis-CI werden in Kapitel 6 unter dem Abschnitt Travis-CI angezeigt.

Kapitel 3

Anforderungsanalyse

Der erste Schritt vor der Entwicklung der Microservice-Architektur besteht darin, das System zu verstehen, das entwickelt werden soll, und alle aus der Perspektive eines Benutzers und aus jener eines Entwicklers erforderlichen Anforderungen zu sammeln und zu analysieren. Es gibt verschiedene Methoden, mit denen sich dies umsetzen lässt. User Stories werden für die Ableitung der Anforderungen beschrieben, um eine genaue Vorstellung davon zu erhalten, was entworfen und schließlich entwickelt werden soll. Dies ermöglicht es auch, die Grenzen des Projekts zu erkennen und die Nutzung der Ressourcen zu optimieren. Das heißt in diesem Fall beispielsweise: die Zeit, die für jede Story/Anforderung aufgewendet wird, und die Fähigkeit, eine Funktion zu entwickeln, ohne gegen eine Wand zu stoßen. Für die Microservice-Architektur ist es von Bedeutung, zunächst zu definieren, welches Feature zu welchem Service gehört. Um dies herauszufinden, müssen die Anforderungen gesammelt und verstanden werden. Ein grobes Verständnis der Verteilung kann dann in einem Diagramm dargestellt werden, das die Aufgaben der einzelnen Services erläutert. Abschließend müssen die folgenden Unterpunkte nacheinander umgesetzt werden:

1. der Prozess, wie man mit der Entwicklung des ersten Services beginnt, ihn mit Docker integriert und in einem Container ausführt, und
2. die Erstellung einer Pipeline zum kontinuierlichen Testen in GitHub mit Travis-CI.
Wenn diese erste Aufgabe erledigt ist, dann
3. kann man mit der Integration des zweiten Dienstes in einem anderen Container beginnen.

3.1 Zielsetzung

„Zu Beginn eines Systementwicklungsprojekts findet die Systemanalyse statt. Diese Aussage wird heutzutage in keinem Projekt mehr angezweifelt. Allerdings gibt es nach wie vor Projektleiter, für die Systemanalyse zwar selbstverständlich ist, denen aber unklar ist, wieso sie gerade in

diese Aktivität viel Zeit und Geld investieren sollen. Trotz allzu bekannter Beispielprojekte aus der Industrie, die mehr denn je die Relevanz der Systemanalyse belegen, haben sie noch immer nicht verstanden, dass gerade die Systemanalyse im Entwicklungsprozess maßgeblich für den Erfolg oder auch Misserfolg eines Projektes verantwortlich ist.“ (Rupp und SOPHISTen 2013, S. 11)

Jedes zweite Projekt schlägt fehlt, da die für eine erfolgreiche Projektplanung zu erfüllenden Ziele unterschätzt werden können. Diese gescheiterten Ziele lassen sich beispielsweise auf ein Überschreiten der geplanten Ressourcen, das Nichteinhalten eines festgelegten Abgabetermins und eine geringe gelieferte Qualität zurückführen.

Das Ziel der Plattform, die auf dieser Architektur aufbaut, ist es, Tennisbegeisterten die Möglichkeit zu geben, miteinander in Kontakt zu treten. Ein Spieler soll in der Lage sein, einen anderen Spieler auf der Plattform zu finden und eine entsprechende Anfrage zu senden. Jeder Spieler kann eine Bewertung zwischen einem Stern und fünf Sternen hinterlassen. Um dieses Ziel erreichen zu können, müssen folgende Funktionalitäten eingestellt werden, die für das Funktionieren der Plattform unerlässlich sind:

1. User Registration, User Login, User Auth
2. Rezensionen/Bewertungen eines Spielers

Nun werden funktionale und nicht-funktionale Anforderungen erhoben, um eine bessere Übersicht einer erfolgreichen Softwareentwicklung zu schaffen. Zu Beginn wird angestrebt, die richtigen Akteure und Use-Cases zu identifizieren. Diese lassen sich wie folgt ableiten.

3.2 Anwendungsumgebung

TODO: Revise this section according to end Die Plattform soll im Rahmen verschiedener Sportaktivitäten definiert werden. In diesem Projekt wird nur die Suche nach Tennisspielern implementiert. Da die Anwendung jedoch auf Microservices basiert, können weitere Dienste von anderen Mitgliedern/Studierenden hinzugefügt werden, die andere Sportarten/Aktivitäten in die Plattform aufnehmen. Dies bedeutet, dass Tennisbegeisterte in der Lage sein sollten, andere Tennisspieler online zu finden und einen Termin für ein gemeinsames Spiel zu vereinbaren. Es gibt hauptsächlich drei interessierte Parteien auf der Plattform: einen Tennisspieler, den Gegner eines Tennisspielers und den Administrator, der die Plattform betreibt. Ein Tennisspieler kann einen Gegner finden und ihm eine Anfrage schreiben, und er kann auch andere Benutzer bewerten. Ein Administrator vermag die Regeln für die Nutzung der Platformdienste festzulegen und unangemessene Benutzer oder Bewertungen zu löschen.

3.3 Rahmenbedingungen

Es gibt zwei wesentliche Grundvoraussetzungen, die vor der Implementierung der Software erfüllt werden müssen. Sie lassen sich in fachliche Kompetenz und technische Voraussetzungen unterteilen.

gen unterteilen. Was den fachlichen Teil betrifft, so wird der Verfasser der vorliegenden Arbeit seinen Betreuer und Professor um Hilfe bitten, um alle Einschränkungen, Beschränkungen und Unklarheiten im Rahmen der Software zu klären. Den technischen Teil betreffend müssen fünf Bedingungen erfüllt werden:

- Die Software wird auf einer Linux-Distribution laufen.
- Sie wird auf Containern laufen.
- Es werden die aktuellsten und beliebtesten Frameworks für die Web-Entwicklung von Backend-Services eingesetzt. Gleichzeitig wird JavaScript für die Entwicklung des Frontends verwendet, da es Cross-Browser-kompatibel ist und über viele leistungsfähige Frameworks und Bibliotheken verfügt, darunter ReactJS, das für die Entwicklung des Frontends verwendet wird.
- Sie wird kontinuierlich mit TRAVIS-CI getestet und in GitHub bereitgestellt.
- Die Website sollte auf allen Rechnern ausgeführt werden können.

3.4 Anforderungen

Die erste Frage, die gestellt werden sollte, ist, wie die Anforderungen am besten gesammelt werden können. Das Sammeln von Anforderungen sollte agil bleiben, um die Möglichkeit zu haben, während der Implementierungsphase weitere Anforderungen zu bearbeiten, zu löschen oder hinzuzufügen, falls erforderlich. Mit dem Ziel, die Komponenten agil zu halten, lassen sich einige Methoden zur Ermittlung dieser Anforderungen verwenden. Wie bereits im vorherigen Kapitel erwähnt, sollten die Anforderungen am besten aus zwei Perspektiven betrachtet werden: zum einen aus der Sicht der Benutzer und zum anderen aus der Sicht des Entwicklerteams, das in diesem Fall nur aus dem Verfasser der Arbeit besteht. Wenn er sich also in die Position des Benutzers versetzt, gelangt er zu den folgenden User Stories. Bevor diese identifiziert werden, gilt es die Stakeholder zu unterscheiden.

3.4.1 Akteure

Wie in Abschnitt 3.1, in der „Zielsetzung“, erwähnt, sollte es zwei Arten von Benutzern und einen Administrator für die Website geben. Eine externe Komponente, die bisher noch nicht erwähnt wurde, stellt der Besucher dar, der der erste ist, der auf die Website zugreift. Um die Akteure zunächst zusammenzufassen, sind sie nachstehend aufgeführt:

1. ein Besucher
2. ein Tennisspieler
3. ein Tennisspieler als Gegner
4. ein Verwalter

3.4.2 Anwendungsfälle und User Stories

Anwendungsfälle

„Ein Anwendungsfall (engl. use case) bündelt alle möglichen Szenarien, die eintreten können, wenn ein Akteur versucht, mit Hilfe des betrachteten Systems ein bestimmtes fachliches Ziel (engl. business goal) zu erreichen. Er beschreibt, was inhaltlich beim Versuch der Zielerreichung passieren kann und abstrahiert von konkreten technischen Lösungen. Das Ergebnis des Anwendungsfalls kann ein Erfolg oder Fehlschlag/Abbruch sein.“ XPhilosoph u. a. 2020

User Stories

In einem Artikel von Bartlett 2016 sind User Stories kurze Beschreibungen von Funktionen, die aus der Sicht des Benutzers erzählt werden. Der Schwerpunkt liegt auf der Frage, warum und wie der Benutzer mit der Software interagiert. Eine User Story ist im Wesentlichen eine High-Level-Definition dessen, was die Software können sollte. Typischerweise wird ein Feedback oder eine Anfrage, die vom Unternehmen oder von dem Endbenutzer stammt, als User Story geschrieben. Eine gute User Story ist in leicht verständlicher Sprache verfasst und beschreibt den Grund sowie den erwarteten Nutzen eines bestimmen Bereichs der Software. Sie folgt normalerweise einer Vorlage wie dieser:

Als <Typ von Benutzer> möchte ich <ein gewünschtes Ergebnis>, damit <ein Grund>. Es folgt ein Beispiel für eine User Story zu der hier vorgestellten Website: Als Besucher möchte ich in der Lage sein, andere Benutzer zu finden und ihre Verfügbarkeit sehen, damit ich ihnen eine Anfrage zum gemeinsamen Tennisspielen schicken kann und wir miteinander spielen können.

Die User Story wird oft von Akzeptanzkriterien begleitet. Diese sind die Grenzen der User Story (Feature) und bestimmen im Wesentlichen, wann die User Story abgeschlossen ist. Die Akzeptanzkriterien bilden auch die Kriterien, anhand derer die Tester ihre Tests schreiben/-durchführen. Sie lassen sich als die funktionalen Anforderungen vorstellen, die eine User Story unterstützen, die Prioritäten bestätigen und die Perspektive des Benutzers in den Ansatz des Entwicklungsteams integrieren.

Auf den folgenden Seiten werden diese User Stories in Tabellenform erstellt und in Epics und Stories kategorisiert. Ein Epic ist im Vergleich zu einer User Story ein großes Werk, das in eine Reihe kleinerer Aufgaben, sogenannte Stories, unterteilt werden kann. Epics hingegen werden von einer Initiative initiiert, die eine Sammlung von Epics ist, die auf ein gemeinsames Ziel hinarbeiten, wie in Abbildung 3.4.1. Anschließend wird aus diesen Stories ein Anwendungsfalldiagramm erstellt.

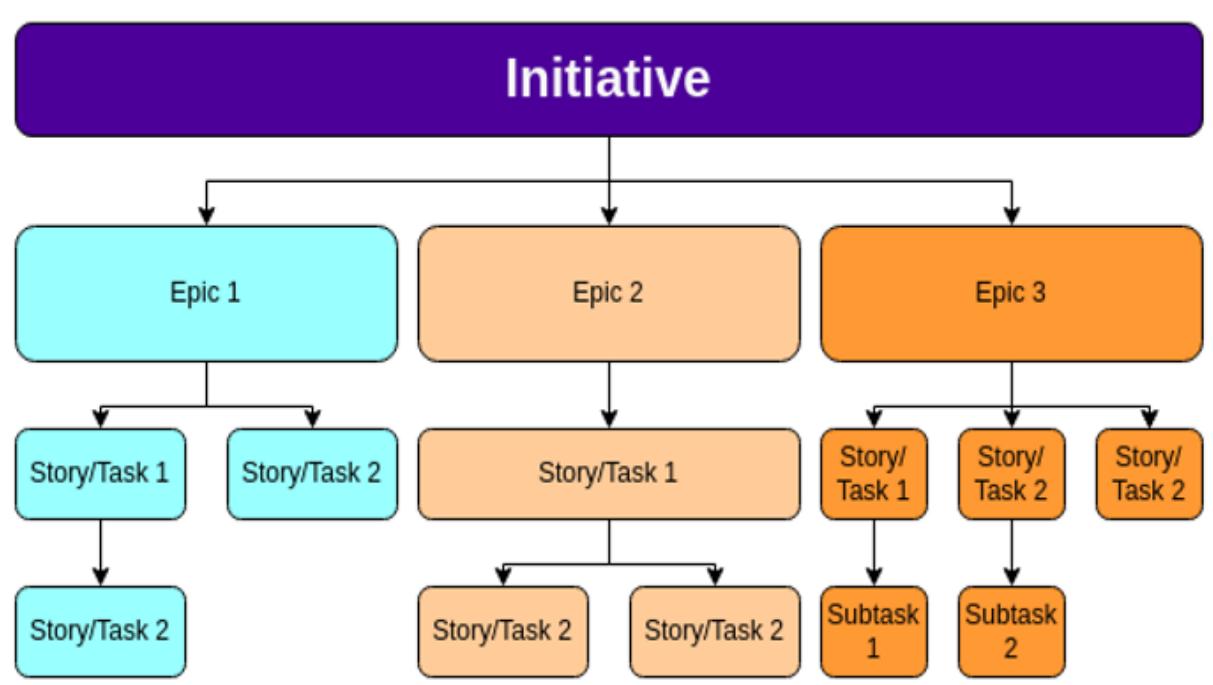


Abbildung 3.4.1: User Stories und Epics. (Quelle: eigene Darstellung)

Anwendungsfälle vs. User Stories

Anwendungsfall	User story
• used in Waterfall methodology	• methodology
• very Detailed	• less Detailed
• It is documented in the FRD	• It is logged onto the JIRA-Tool
• time consuming to create	• relatively faster
• Format: <ol style="list-style-type: none">1. Use Case name2. description3. pre-condition4. post-condition5. basic path6. alternative path7. exception path	• Format: <ol style="list-style-type: none">1. As a ...{WHO}, I want ...{WHAT}, So that ...{WHY}2. acceptance criteria3. supporting models

Tabelle 3.1: Use Case Vs. User story

Da es nun möglich ist, zwischen einem Epic und einer Initiative zu unterscheiden, lassen sich die Aufgaben entsprechend kategorisieren, und mit Hilfe von Tabelle 3.1 werden die ersten Initiativen und deren Epics abgeleitet.

Rolle	Wunsch/Ziel	Kategorie
Als Besucher	will ich die Möglichkeit haben, mich im Portal zu registrieren, damit ich es benutzen kann.	Epic
Als Besucher	will ich mich über eine private E-Mail-Adresse registrieren können.	Story
Als Besucher	will ich mich mit meinem Gmail-Konto über Google registrieren können, damit ich meine Daten nicht nochmal eingeben muss.	Story

Tabelle 3.2: Initiative 1: Epic - Benutzerregisterierung

Rolle	Wunsch/Ziel	Kategorie
Als Tennisspieler-/gegner	will ich mich ohne Anstrengung einloggen können.	Epic
Als Tennisspieler-/gegner	will ich die Option nutzen können, mich mit meiner privaten E-Mail-Adresse einzuloggen.	Story
Als Tennisspieler-/gegner	will ich mich mit anderen Optionen einloggen können, wie Gmail oder Facebook.	Story
Als Tennisspieler-/gegner	will ich bei der falschen Angabe eines Passworts das Passwort zurücksetzen können.	Story

Tabelle 3.3: Initiative 1: Epic - Benutzeranmeldung

Rolle	Wunsch/Ziel	Kategorie
Als Tennisspieler-/gegner	will ich Tennisspieler nach einer Kategorie/nach Eigenschaften suchen können.	Epic
Als Tennisspieler	will ich über einen Button auf die/den gefundene(n) Tennisgegner klicken können, um Informationen zu ihnen einzusehen.	Story
Als Besucher	will ich beliebte und neue Benutzer auf dem Portal sehen.	Story
Als Tennisspieler können	will ich auch empfohlene Benutzer auf dem Portal sehen können.	Story

Tabelle 3.4: Initiative 1: Epic - Benutzerkategorien und Suche

Rolle	Wunsch/Ziel	Kategorie
Als Tennisspieler	will ich mein eigenes Profil verwalten können.	Epic
Als Tennisspieler	will ich ein Profilfoto hochladen können.	Story
Als Tennisspieler	will ich den Beschreibungstext über mich bearbeiten können.	Story
Als Tennisspieler	will ich meine Profilinformationen wie Verfügbarkeit und Stärke bearbeiten können	Story
Als Tennisspieler	will ich einen freien Timeslot hinzufügen können	Story
Als Tennisspieler	will ich einen freien Timeslot löschen können	Story
Als Tennisspieler	will ich einen freien Timeslot bearbeiten können	Story
Als Tennisspieler	will ich die Bewertungen über mich anzeigen lassen können	Story
Als Tennisspieler	will ich auf eine Bewertung antworten können	Story

Tabelle 3.5: Initiative 1: Epic - Profilverwaltung

Rolle	Wunsch/Ziel	Kategorie
Als Tennisspieler-/gegner	will ich Bewertungen hinterlassen und Bewertungen von anderen beanworten können	Epic
Als Tennisspieler	will ich eine Bewertungen bearbeiten können	Story
Als Tennisspieler	will ich Bewertungen löschen können	Story
Als Tennisspieler	will ich alle Bewertungen von Tennisspieler-gegnern über einen Klick auf ihr Profil einsehen können	Story
Als Tennisspieler-/gegner	will ich eine Rezension und eine Bewertung hinzufügen können	Story
Als Tennisspieler-/gegner	will ich vom Tennisspieler bewertet werden	Story

Tabelle 3.6: Initiative 2: Epic - CRUD von Bewertungen

Rolle	Wunsch/Ziel	Kategorie
-------	-------------	-----------

Als Tennisspieler	will ich die freien Timeslots eines ausgewählten Tennisgegners sehen können	Epic
Als Tennisspieler	will ich einen freien Timeslot eines Gegners auswählen können	Story
Als Tennisspieler	will ich einen gewählten Timeslot wieder löschen können	Story

Tabelle 3.7: Initiative 3: Epic - Verwaltung von freien Timeslots

Rolle	Wunsch/Ziel	Kategorie
Als Tennisspieler-/gegner	will ich eine Anfrage verwalten können	Epic
Als Tennisspieler	will ich eine Nachricht bzw. eine Anfrage an einen Tennisspielergegner senden können	Story
Als Tennisspieler	will ich meinen Namen und meine Handynummer für weitere Kontaktmöglichkeiten hinterlassen können	Story
Als Tennisspieler	will ich über einen Button eine Nachricht mit allen relevanten Angaben und zudem Anfragen schicken können.	Story
Als Tennisspieler-/gegner	will ich alle empfangenen Anfragen anzeigen können	Story
Als Tennisspieler-/gegner	will ich eine Anfrage löschen können	Story
Als Tennisspieler-Gegner	will ich die Anfragen von anderen beantworten können	Story

Tabelle 3.8: Initiative 4: Epic - Anfragnverwaltung

3.4.3 Use-Case-Diagramm

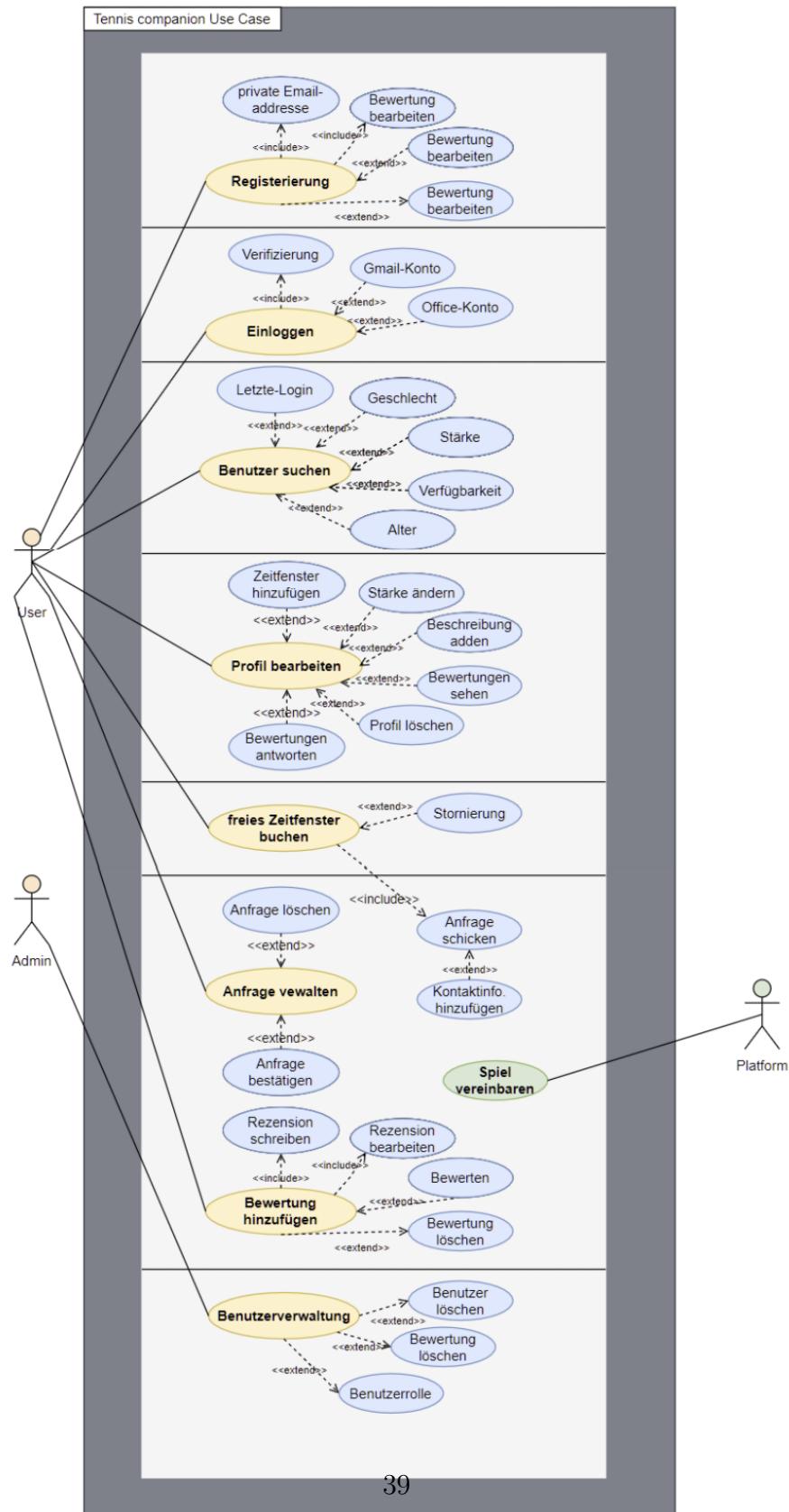


Abbildung 3.4.2: User-Case-Diagramm

3.4.4 MoSCoW-Methode

Die Buchstaben in MoSCoW haben die folgende Bedeutung:

- M steht für Must haves. Dies sind nicht verhandelbare für das Team obligatorisch Produktanforderungen.
- S steht für Should haves. Es handelt sich um wichtige Initiativen, die zwar nicht unerlässlich sind, aber einen erheblichen Mehrwert bieten.
- C steht für "könnte man haben". Dies sind Initiativen, die man gerne hätte, deren Weglassen aber nur eine geringe Auswirkung hat.
- W steht für "will nicht haben" – Initiativen, die in diesem speziellen Zeitrahmen nicht vorrangig sind.

Gemäß MoSCoW werden die Aufgaben nach ihrer Relevanz geordnet und auf der Grundlage der für das Funktionieren des Systems wichtigsten und bedeutendsten Aufgaben abgeschlossen. In der nachstehenden Tabelle sind die Aufgaben entsprechend verteilt.

Must-Have	Should-Have	Could-Have
<ol style="list-style-type: none"> 1. Projekt 1: Frontend-Service 2. Projekt 2: Use-Management -Service + Datenbank + JWT-Authentifizierung: 3. Project 3: Review-Service + Datenbank 4. Verkapselung von Services in Containern 5. Konfiguration von RabbitMQ 6. erste erfolgreiche Kommunikation zwischen dem zweiten und dem dritten Projekt über RabbitMQ 7. Konfiguration von NGINX-API-Gateway 	<ol style="list-style-type: none"> 1. Erstellung eines ER-Diagramms. 2. Testen des zweiten Projekts 3. Testen des dritten Projekts 4. Continuous Integration mit Travis-CI. 5. Code auf GitHub veröffentlichen. 	<ol style="list-style-type: none"> 1. Projekt 4: Erstellung von Timeslot-Service innerhalb von Projekt 2 aus Gründen der Vereinfachung 2. Projekt 5: Erstellung eines Anfrage-Services innerhalb von Projekt 2 aus Gründen der Vereinfachung 3. Abtrennung des vierten Projekts und Übertragen in einen neuen Service + neue Datenbank 4. Abtrennung des fünften Projekts und Übertragen in einen neuen Service + neue Datenbank 5. Testen des vierten Projekts. 6. Testen des fünften Projekts.

Tabelle 3.9: MoSCoW Methode

Die „Won’t-Haves“ werden zunächst nicht definiert, aber wenn sich etwas ändert, wird eine entsprechende Anpassung vorgenommen.

Kapitel 4

Systementwurf und Konzeption

In diesem Kapitel werden alle im vorherigen Kapitel zusammengetragenen Anwendungsfälle und Anforderungen grafisch dargestellt. Ziel ist es, zu zeigen, wie die verschiedenen Systeme, d. h. die Dienste, unter Verwendung vereinfachter Designstrukturen miteinander kommunizieren werden. Außerdem wird ein Mock-up erstellt, um das Endergebnis, das Besuchern der Website ersichtlich wird, zu zeigen. Dies ermöglicht es, eventuellen Änderungsbedarf an den Anforderungen und Funktionen zu erkennen, bevor mit der Umsetzung begonnen wird. Nachdem in diesem Kapitel eine klare Visualisierung des Systems erstellt wurde, werden im nächsten Kapitel die einzelnen Elemente nacheinander durchgegangen.

4.1 Entwurf Systemarchitektur

In diesem Abschnitt wird ein grobes Verständnis des gesamten Systems geschaffen. Zudem gilt es eine kurze Beschreibung vorzunehmen.

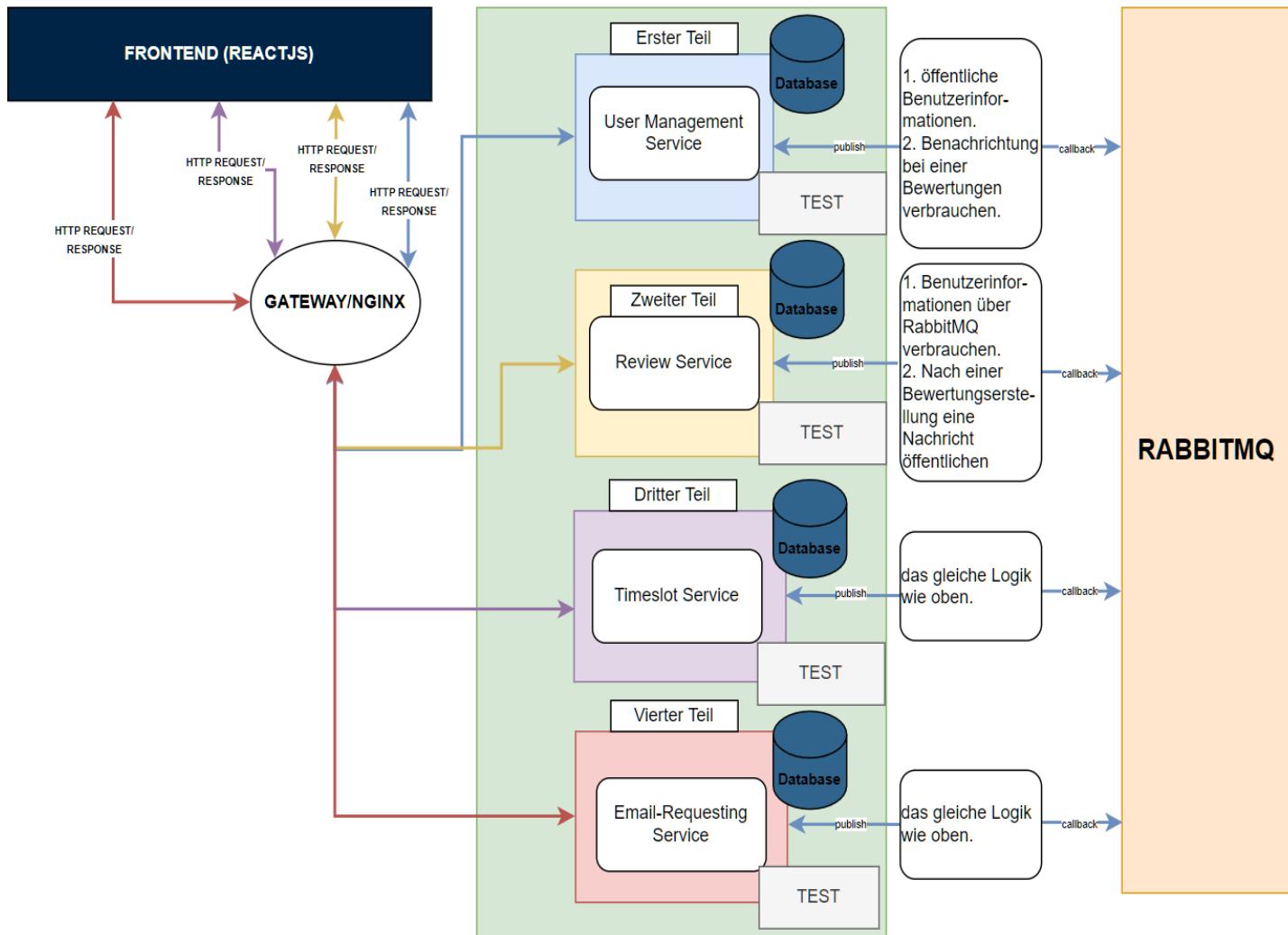


Abbildung 4.1.1: Grobe Übersicht des gesamten Architektur (Quelle: eigene Darstellung)

Wie in den vorangegangenen Kapiteln beschrieben, wird es verschiedene Services geben. Diese werden der Einfachheit halber hauptsächlich in Form einer monolithischen Architektur konzipiert. Wie die obige Abbildung zeigt, gibt es im Backend vier Apps. Nach der vollständigen Entwicklung der Apps in den „Must-Haves“ wird angestrebt, diese zu trennen, mit dem Ziel, eine funktionale Abgrenzung und eine erfolgreiche Kommunikation zwischen den abgetrennten Apps zu gewährleisten, um den ersten Schritt in Richtung einer Microservice-Architektur zu gehen. Aus Zeitgründen und aufgrund der Komplexität des Projekts kann es sein, dass einige Apps übrig bleiben, die nicht entwickelt werden.

Neben dem Backend gibt es zwei weitere bedeutende Komponenten. Die eine ist der Nginx, der als API-Gateway zwischen dem Frontend und den verschiedenen Services eingesetzt werden soll. Darüber hinaus werden die Services über RabbitMQ miteinander kommunizieren können. Dies ermöglicht es jedem Service, unabhängig von den anderen zu erfahren, ob sich in einem der Services etwas geändert hat, indem er sich bei diesen abonniert. Das heißt, wenn beispielsweise ein Benutzer angelegt wird, veröffentlicht der Benutzerverwaltungsdienst eine Nachricht an RabbitMQ, und die anderen Services wie der Review-Service konsumieren diese Nachricht bei Bedarf.

4.2 Abgrenzung

Bevor mit der Implementierung des gesamten Systems, das in Abbildung 4.1 dargestellt ist, begonnen wird, gilt es den Umfang dessen zu bestimmen, was innerhalb der begrenzten Projektbearbeitungszeit fertiggestellt werden kann. Aus diesem Grund werden zu Beginn nur die ersten beiden in Abbildung 4.1 sichtbaren Teile getrennt. Die Teile 3 und 4 verbleiben im ersten Teil, bis eine erste Konfiguration erfolgreich durchgeführt wurde und genügend Zeit vorhanden ist, um die restliche Teile bzw. Services sowohl zu implementieren als auch zu trennen.

4.3 Festlegung der Technologien

Im Rahmen dieser Arbeit spielt eine Reihe von Technologien eine wesentliche Rolle für die verschiedenen Komponenten des Programms . Mindestens zehn verschiedene Technologien müssen beherrscht werden, darunter Django, ReactJS, RabbitMQ, PostgreSQL, NGINX, Dockers, Github, Travis und Rest APIs.

Server:

Die Server werden auf Containern laufen und über die Netzwerk-Container miteinander in Kontakt treten. Mit NGINX werden die Anfragen dann entsprechend weitergeleitet.

Container-basierte Software

Mit Hilfe von Docker kann das Programm auf verschiedenen Computern ausgeführt werden, ohne dass die unterschiedlichen Abhängigkeiten, Betriebssysteme usw. berücksichtigt werden müssen. Diese Containerisierung verschiedener Dienste ermöglicht eine konsistente Bereitstellung sowie die Konfiguration und Automatisierung sich wiederholender Aufgaben, die sich mithilfe des Docker Composer optimieren lassen.

Platform:

Ausgehend von den im vorherigen Kapitel genannten Anforderungen soll zunächst mit Python ein Portal für Tennisbegeisterte entwickelt werden. Ein Teil der Architektur wird die

Möglichkeit eröffnen, dass weitere Entwickler eine andere Programmiersprache verwenden, um weitere Funktionen zu entwickeln, ohne den Kern der Anwendung zu beeinträchtigen. Einer der wichtigsten Services im Rahmen dieser Arbeit wird der User-Management-Service sein, da er von allen anderen Services benötigt wird. Somit ist es von Bedeutung, ihn so lose wie möglich zu koppeln, damit andere Services bei Bedarf wichtige Benutzerinformationen von ihm erhalten können. Dies ermöglicht es, andere IDs verschiedener Entitäten/Module entsprechend zuzuordnen.

Ein Meilenstein neben der Abtrennung eines erfolgreichen User-Management-Services ist es, die restlichen Services innerhalb eines Projekts in einer monolithischen Angelegenheit zu entwickeln, um eine funktionierende Anwendung hervorzubringen. Im weiteren Verlauf werden andere Services wie der Review-Service und der Rest getrennt. Das Trennen von mindestens einem Service hat Priorität dahingehend, die anfängliche Funktionalität des Systems und die Kommunikation mit RabbitMQ unter Verwendung der Microservice-Architektur zu testen.

Django:

Das Hauptziel von Django besteht darin, die Erstellung komplexer, datenbankgestützter Websites zu erleichtern. Das Framework betont die Wiederverwendbarkeit und die „Steckbarkeit“ von Komponenten. Hinzu kommen die Aspekte weniger Codes, geringe Kopplung, schnelle Entwicklung und der Grundsatz „Wiederhole dich nicht“ (Djangodocu 2018).

Model:

Laut Djangodocu n.d. ist jedes Modell in Django eine Python-Klasse, die die UnterkLASSE django.db.models.Model bildet. Jedes Attribut des Modells repräsentiert ein Datenbankfeld. Mit all dem gibt Django eine automatisch generierte Datenbankzugriffs-API an die Hand, die es ermöglicht, Abfragen zu kreieren.

View:

In dieser Schicht wird ReactJS verwendet und seine Fähigkeit, innerhalb der View-Layer HTML sowie einen dynamischen Code zu erstellen, voll ausgenutzt. CSS und Bootstrap können auch hinzugefügt werden, um das Design in jedem Punkt anzupassen. Dabei wird berücksichtigt, dass das Template, das die View-Layer in Django darstellt, nicht verwendet wird.

4.4 Model-Layer

In der Model-Layer gibt es vier Initiativen – User-Management, Review-Management, E-Mail-Request-Management und Timeslot-Management –, wie im vorherigen Kapitel erwähnt. Die Modelle und Attribute für jede Initiative werden in diesem Unterkapitel vorgestellt, bevor mit dem Teil der Implementierung begonnen wird.

4.4.1 Initiative 1: User-Management

Hier sollen die Registrierung sowie das Einloggen in das Portal ermöglicht werden. Aus diesem Grund wird es für diesen Teil zwei Apps bzw. Epics geben. Eine ist der User, auch mit `manage_user` bezeichnet, und die andere Profiles.

User

In User-Model werden Attributen mit Aussagefähigen Namen wie `username`, `first_name`, `last_name`, `email`, `is_staff`, `is_active`, `data_joined`, `is_signed` versehen. Wobei das Attribut `is_staff` trägt dazu bei, dass die Benutzer die richtigen Berechtigungen haben. Wenn es True ist, bedeutet dies, dass der Benutzer ein Admin ist und False, dass er ein normaler aktiver Benutzer ist.

Profile

Im User-Modell werden Attribute mit aussagekräftigen Namen wie `username`, `first_name`, `last_name`, `email`, `is_staff`, `is_active`, `data_joined` und `is_signed` versehen. Das Attribut `is_staff` trägt dazu bei, dass die Benutzer die richtigen Berechtigungen haben. Wenn es true ist, bedeutet dies, dass der Benutzer den Status eines Admins hat. False bringt zum Ausdruck, dass er ein normaler aktiver Benutzer ist.

- Gender: Hier wird vermerkt, ob der User männlich oder weiblich ist, bzw. ein anderes Geschlecht hat.
- Skill-Level: Die Skill-Level werden in fünf Stufen unterteilt: Beginner, Advanced Beginner, Competent, Advanced und Expert.
- Game-Type: Die Game-Level werden in fünf Stufen unterteilt: Badminton, Paddle Tennis, Squash, Table Tennis und Tennis.

4.4.2 Initiative 2: Review-Management

Da dies der erste Service ist, der in ein anderes Projekt ausgegliedert wird, gilt es die richtigen Attribute im Modell dieses Services zu wählen, damit der Service oder das Modell die richtige Review-ID der richtigen User-ID zuordnen kann, die vom User-Management-Service konsumiert wird.

Rater

Somit ist es von hoher Bedeutung, in diesem Service eine zusätzliche Klasse bzw. ein zusätzliches Modell mit der Bezeichnung „Rater“ zu erstellen. Dies ermöglicht es dem Service, den Benutzernamen und die ID, die er von einem anderen Service konsumiert, in seiner eigenen unabhängigen Datenbank zu speichern, wo der Benutzername anschließend mit einer entsprechenden Bewertungs-ID verknüpft werden kann. Diese Klasse hat die folgenden Attribute: `pkid`, `id`, `username`, `nr_rated_users`, `is_signed`, `created_at` und `updated_at`.

Rating

Neben dem „Rater“-Modell wird es auch ein „Rating“-Modell geben. Es besteht aus den Attributen `id`, `range`, `rater`, `rated_user`, `rating` und einem `comment`. Die Attribute `rater` und `rated_user` werden als Fremdschlüssel deklariert und übernehmen das Rater-Modell, sodass jede `rating_id` einer `rater_id` zugeordnet wird. Der Fremdschlüssel in Django bietet eine Many-To-One-Beziehung, indem er dem lokalen Modell eine Spalte hinzfügt, die den entfernten Wert enthält. Standardmäßig zielt ForeignKey auf den pk des entfernten Modells ab. Aber dieses Verhalten kann mit dem Argument `to_field` geändert werden. Eines dieser Attribute, mit `range` bezeichnet, wird es dem Benutzer ermöglichen, zwischen verschiedenen Optionen zu wählen, wie im Folgenden dargestellt:

- Range: Die Range wird in fünf Stufen unterteilt: Poor, Fair, Good, Very Good, Excellent.

4.5 View-Layer

In diesem Unterkapitel werden die Epics des vorherigen Kapitels nochmal herangezogen, um den ersten Ansichts-Prototyp für die Website zu erstellen.

Benutzerregistrierung

In der Abbildung zur Benutzerregistrierung ist zu sehen, dass ein Benutzer all seine Informationen eingeben kann. Wenn er anschließend auf die Taste „Create my free account“ klickt, wird ein Benutzer im User-Management-Servie erstellt und erhält einen Nicht-Administrator-Zugang zu der Website. Nachdem sich der Benutzer erfolgreich registriert hat, sollte er auf die Anmeldeseite weitergeleitet werden.

Tennis Companion Registration Form

First Name <input type="text" value="bruce@wayne.com"/>	Last Name <input type="text" value="bruce@wayne.com"/>
Email <input type="text" value="bruce@wayne.com"/>	Country <input style="width: 100px; height: 20px; border: 1px solid #ccc; border-radius: 5px; padding: 5px; margin-bottom: 5px;" type="text" value="Germany"/> <div style="border: 1px solid #ccc; width: 20px; height: 15px; background-color: #f0f0f0; position: absolute; right: -10px; top: 5px;"></div>
Phone number <input type="text"/>	Postcode <input type="text"/>
City <input type="text"/>	State <input type="text"/>
Gender <input type="text"/>	Birth Date <input style="width: 80px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 5px; margin-right: 10px;" type="text"/> Month <input style="width: 20px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 5px; margin-right: 10px;" type="text"/> Day <input style="width: 20px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 5px; margin-right: 10px;" type="text"/> Year <input style="width: 20px; height: 25px; border: 1px solid #ccc; border-radius: 5px; padding: 5px;" type="text"/>
Skill Level <input style="width: 250px; height: 30px; border: 1px solid #ccc; border-radius: 5px; padding: 5px; margin-bottom: 5px;" type="text"/> <div style="border: 1px solid #ccc; width: 20px; height: 15px; background-color: #f0f0f0; position: absolute; right: -10px; top: 5px;"></div>	Game Type <input style="width: 250px; height: 30px; border: 1px solid #ccc; border-radius: 5px; padding: 5px; margin-bottom: 5px;" type="text"/> <div style="border: 1px solid #ccc; width: 20px; height: 15px; background-color: #f0f0f0; position: absolute; right: -10px; top: 5px;"></div>
Password <input type="text" value="at least 8 characters"/>	Confirm Password <input type="text" value="Confirm password"/>

[Create my free account](#)

[Sign up with Google](#)

Abbildung 4.5.1: Mockup – Registrierungsfenster

Benutzeranmeldung

Hier wird ein Benutzer seine registrierte E-Mail-Adresse und sein Passwort verwenden. Mit diesen beiden Daten sendet der Server dann ein „Access“- und „Refresh“-Token, das für die Sitzung während der Anmeldezeit auf der Website verwendet wird. Sobald er sich abgemeldet hat, wird ein neues Token benötigt, um eine neue Sitzung eröffnen zu können.

The mockup shows a user interface for logging in to the 'Tennis Companion' application. At the top, there is a header bar with the 'Tennis Companion' logo on the left and navigation links 'User', 'Home', 'My Profile', and 'About Us' on the right. Below the header, the main content area has a title 'Tennis Companion'. It contains two input fields: one for 'Email' with the placeholder 'bruce@wayne.com' and another for 'Password' with the placeholder 'at least 8 characters'. Below the password field is a link 'Forgot password?'. A large black button in the center contains the text 'Log in to Tennis Companion'. At the bottom, there is a 'Sign in with Google' button featuring the Google logo and the text 'Sign in with Google'.

Abbildung 4.5.2: Mockup – Einloggen eines Benutzers

Hauptseite

Auf dieser Seite werden alle registrierten Benutzer angezeigt. Sie können nach Verfügbarkeit gesucht und nach Beliebtheit sowie aktuellsten Registrierungen auf dem Portal angezeigt werden. Es wird auch möglich sein, anhand eines Namens nach einem bestimmten Benutzer zu suchen.

Wenn auf eines der Benutzerprofile geklickt wird, sollte das in Abbildung 4.4.5 ersichtliche Mockup angezeigt werden. „Andrew Wayne“ wird als Beispielbenutzer verwendet.

In der Kopfzeile (engl.: header) wird beim Klicken der User-Taste dem Anmeldestatus des Benutzers entsprechend eine Auswahlliste angezeigt: Wenn der Benutzer angemeldet ist, wird Abbildung 4.4.4 (a) angezeigt, andernfalls Abbildung 4.4.4 (b).

The mockup displays a grid of user profiles. At the top left is the 'Tennis Companion' logo. To its right is a navigation bar with links: User, Home, My Profile, and About Us. Below the navigation is a search bar with a magnifying glass icon and the placeholder text 'Search for a competitor'. The main content area contains a 3x3 grid of user cards. Each card features a circular profile picture of Shrek, followed by the username, a short description, the number of reviews, and a five-star rating. The first row shows three cards: 'Shrek1' (description: 'a professional Tennis player, whose playing since 2010 with the best Tennis leagues'), 'Username' (empty), and 'Username' (empty). The second row shows three cards: 'Reviews' (empty), 'Reviews' (empty), and 'Reviews' (empty). The third row shows three cards: 'Reviews' (empty), 'Reviews' (empty), and 'Reviews' (empty). The entire grid is set against a light gray background.

Abbildung 4.5.3: Mockup - Hauptseite und der Suche nach Benutzern

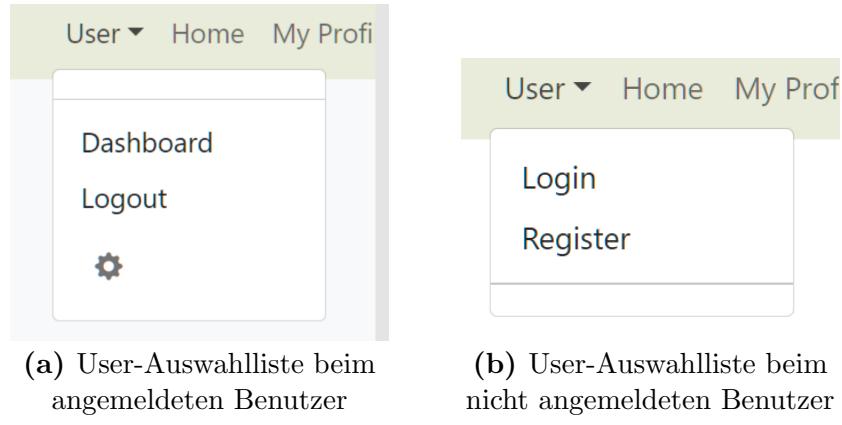


Abbildung 4.5.4: Auswahlliste des Benutzers entsprechend dem Anmeldungsstatus

Profil-Management

In diesem Abschnitt gibt es verschiedene Mocks, darunter die Profilseite eines bestimmten Benutzers im Portal und das Profil des angemeldeten Benutzers.

Außerdem haben die Benutzer die Möglichkeit, ihre Profile zu bearbeiten. Die Seite, auf der sich dies umsetzen lässt, wird ebenfalls dargestellt.

Am Ende des Abschnitts „Profil-Management“ werden weitere Mocks gezeigt, um zu veranschaulichen, welche Seiten ein Administrator sehen kann.



Andrew Wayne (2.5)



Available: Yes/No

Profile Information

Lives in: Berlin, MA

[Contact To Play](#)[Block](#)

Gender: Male

We'll send an Email.

Age: 21

Skill Level: 2.5 Advanced Player

 [Add to My Favorite list](#)

4.89 9 Reviews



Lennart

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.



Lennart

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.



Lennart

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.



Lennart

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.



Lennart

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.



Lennart

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.



Lennart

December 21

I played with Andrew, and he is a cool guy and



Lennart

December 21

I played with Andrew, and he is a cool guy and

Abbildung 4.5.5: Mockup - Profilseite eines Benutzers

Hier gibt es drei bedeutende Felder, wie in Abbildung 4.4.5 zu sehen ist. Das erste Feld zeigt das Bild des Benutzers und ob er für ein Spiel zur Verfügung steht. Das zweite Feld, „Profile Information“ sollte es ermöglichen, den Benutzer durch Klicken auf die Taste „Contact To Play“ zu kontaktieren. Darüber hinaus können die Profilinformationen, wie Alter, Region und Spielstärke, angezeigt werden. Außerdem sollten unter dem Feld zu den Profilinformationen alle Bewertungen aufgeführt werden, die zu diesem Benutzer verfügbar sind.

**Sakhr Al-absi (1.0)**

Availability: yes/no

Profile Settings

Profile Status:

You may temporarily make yourself unavailable to play tennis by setting the profile status to "private".

Your Email: bruche@wayne.com  ChangePassword: *****  Change[Edit profile](#)[Become a Coach](#)

★ 4.89 9 Reviews

**Andrew**

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.

**Lennart**

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.

**Lennart**

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.

**Lennart**

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.

**Lennart**

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.

**Lennart**

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.

**Lennart**

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.

**Lennart**

December 21

I played with Andrew, and he is a cool guy and would love to have a revenge match.

Abbildung 4.5.6: Mockup - Meine Profilseite

Aus Abbildung 4.4.6 geht hervor, dass es die Möglichkeit gibt, einige Änderungen auf der Seite vorzunehmen. Zum Beispiel kann der angemeldete Benutzer ein neues Bild hochladen oder ein bestehendes bearbeiten. In dem folgenden Feld lässt sich das Profil bearbeiten, indem beispielsweise der Name und das Passwort geändert werden. Unten auf der Seite lassen sich die für diesen Benutzer hinterlassenen Bewertungen anzeigen, wie auch in Abbildung 4.4.5 ersichtlich.

Admin-Seite

Wenn der Benutzer ein Administrator ist, stehen andere Optionen zur Verfügung, die auch für einen normalen Benutzer verfügbar sind. Neben diesen wird der Administrator eine andere Dashboard-Seitenleiste sehen können. Zu diesem Dashboard zählt die Option „Users of Tennis Companion“, wie in Abbildung 4.4.7 gezeigt.

Auf dieser Seite werden alle registrierten Benutzer von „Tennis Companion“ in einer Tabelle aufgelistet. Zu den angezeigten Informationen zählen: der Benutzername, das Erstellungsdatum und die Aktion, die der Administrator durchführen möchte, das heißt, entweder alle Bewertungen eines bestimmten Benutzers zu überprüfen oder ihn zu löschen.

The mockup shows a web interface for an administrator. At the top, there is a header bar with the title "Tennis Companion" on the left and navigation links "Admin", "Home", "My Profile", and "About Us" on the right. Below the header is a search bar with the placeholder "Search for a competitor". On the left, there is a sidebar menu titled "Dashboard" containing links for "Dashboard", "Users of Tennis Companion", "Change Password", and "Logout". The main content area is titled "Users" and displays a table with the following columns: "User name", "Created at", and "Action". The table lists eight entries, all with the user name "andrew@gmail.com", and each entry has two buttons in the "Action" column: a green "Reviews" button and a red "Delete" button. The table rows are separated by horizontal lines.

Users		
User name	Created at	Action
andrew@gmail.com		<button>Reviews</button> <button>Delete</button>

Abbildung 4.5.7: Mockup - Admin Dashboard und alle registrierten Benutzern

🔍

Dashboard				
Dashboard Users of Tennis Companion Change Password Logout				
User Reviews				
Review ID	User Reviewed	Created At	Action	
andrew	zacki		Reviews	Delete
test user1	test user2		Reviews	Delete
test user 2	test user 3		Reviews	Delete
test 3	zacki		Reviews	Delete
test user 4	andrew		Reviews	Delete
test user 5	test user 4		Reviews	Delete
test user 6	test user 5		Reviews	Delete
test user 7	test user6		Reviews	Delete

Abbildung 4.5.8: Mockup - Admin Dashboard und alle Bewertungen einer Benutzer-ID

Wenn die Taste „Reviews“ geklickt wird, erscheint die Abbildung 4.4.8. Auf dieser Seite werden alle bewerteten Benutzer dieses spezifischen Benutzers aufgelistet. Der Administrator kann die Bewertungen überprüfen und bei Bedarf eine löschen. In der Tabelle sind vier Zeilen zu sehen: Review ID, User Reviewed, Created At und Action.

Sobald beispielsweise eine der Bewertungen angeklickt wird, erscheint die Information oder die in der Abbildung 4.4.9 ersichtliche Seite.

The image shows a web-based admin dashboard for a tennis companion application. At the top, there is a header bar with the title "Tennis Companion" on the left and navigation links "Admin", "Home", "My Profile", and "About Us" on the right. Below the header is a search bar with a magnifying glass icon and the placeholder text "Search for a competitor".

The main content area is divided into two sections. On the left is a sidebar titled "Dashboard" containing links for "Dashboard", "Users of Tennis Companion", "Change Password", and "Logout". On the right is a table titled "User Reviews" with columns for "Review ID", "Comment", "Rating", and "Action". The table contains one row of data:

Review ID	Comment	Rating	Action
andrew	zacki	★★★★★	Reviews Delete

Abbildung 4.5.9: Mockup - Mockup - Admin Dashboard und die Bewertung-ID einer spezifischen Benutzer-ID

4.6 Controller-Layer

In diesem Abschnitt werden die bedeutendsten APIs dargestellt. Dies geschieht in Abhängigkeit davon, zu welcher Tabelle sie gehören. Außerdem werden alle wichtigen Funktionen erläutert. Des Weiteren wird die Verwendung von RabbitMQ demonstriert und aufgezeigt, wie der Users-Management-Service und der Review-Service über das „Advanced Message Queuing Protocol“ (AMQP) Nachrichten untereinander austauschen.

Nr.	CRUD ¹	Endpoint	Permission	Task	EPIC
1	POST	users/api/register/	isNAAuth ²	Benutzerregisterierung	User registration
2	POST	users/api/token/	isAuth ³	refresh token zurückgeben	User-Login
3	POST	users/api/token/refresh/	isAuth	access & refresh tokens zurückgeben	User-Login
4	POST	users/api/google/	isNAAuth	Gmail Benutzerauthentifizierung	User registration
5	POST	users/api/logout/blacklist/	NA	aktives Token zur Blackliste senden	User-Login
6	POST	users/api/v1/users/logout/	isAuth	Benutzerabmeldung	User-Login
7	GET	users/api/filter-available-users/	isNAAuth	Profile nach Verfügbarkeit filtern	Benutzerkategorien und Suche
8	GET	users/api/filter-by-id/	isNAAuth	Profile nach ID filtern	Benutzerkategorien und Suche
9	GET	users/api/registered-users/	isAuth	alle Profile abrufen	Benutzerkategorien und Suche
10	GET	users/api/popular-players/	isNAAuth	Profile von beliebten Benutzern abrufen	Benutzerkategorien und Suchet
11	GET	users/api/latest-players/	isAuth	Profile von neusten Benutzern abrufen	Benutzerkategorien und Suche
12	GET	users/api/recommended-players/	isAAAuthz	Profile von empfohlenen Benutzern abrufen	Benutzerkategorien und Suche

¹CRUD: Create, Read, Update, Delete

²Benutzer braucht keine Authentifizierung

³Benutzer braucht eine Authentifizierung

13	POST	users/api/update-user-availability/<str:pk>/	isAuth	Aktualisierung Verfügbarkeit Benutzer-ID	der einer	Benutzerkategorien und Suche
14	GET	users/api/v1/my-profile/	isAuthz ⁴	eignes Profil abrufen		Profil-Management
15	GET	users/api/user-details/<int:pk>/	isAuth	Benutzerdaten einer Benutzer-ID		Profil-Management
16	GET	/users/api/reset/done/	NA	response: Password reset complete		Profil-Management
17	POST	users/api/reset-password/	NA	Passwordrücksetzung		Profil-Management
18	POST	users/api/reset/<str:uidb64>/<token>/	NA	Bestätigung der Passrücksetzung		Profil-Management
19	PATCH	users/api/profile/update/<str:username>/	isAuthz	Ein Profilattribut aktualisieren		Profil-Management
20	DELETE	users/api/delete-user/	isAuthz	Eigenes Profil löschen		Profil-Management
21	DELETE	users/api/delete-user/<str:username>/	isAAuthz	Einen Benutzer über einen Benutzernamen löschen		Profil-Management

Tabelle 4.1: Initiative 1 - User-Management-Service API's

In Tabelle 4.1 sind alle APIs zum Abrufen, Erstellen, Aktualisieren, Bearbeiten und Löschen von Benutzern aufgeführt, jeweils mit dem entsprechenden Endpoint und der Aufgabe des jeweiligen Aufrufs.

Initiative 1: User-Management

Es werden verschiedene Bibliotheken verwendet und die Epics einer nach dem anderen abgearbeitet, um das Ziel der Verwaltung eines Benutzers zu erreichen – von der Erstellung eines neuen Kontos im Tennis-Companion-Portal bis zu dessen Löschung. Dies wird in den folgenden Abschnitten kurz erläutert.

EPIC 1: Benutzerregisterierung

Bei der Registrierung mit einer privaten E-Mail-Adresse wird die Profil-Entität mit der Benutzer-Entität verknüpft, sodass bei der Erstellung eines Benutzers gleichzeitig eine Profil-Entität für diesen Benutzer in die Datenbank aufgenommen wird.

⁴Benutzer ist authentifiziert und autorisiert

In den Rumpf der Anfrage müssen alle Profildaten mit der E-Mail-Adresse, dem Passwort und dem Benutzernamen eingegeben werden.

Wenn die Registrierung über die Google-Option ausgeführt wurde, werden alle Daten automatisch von Google übernommen und in der Datenbank gespeichert.

EPIC 2: Benutzeranmeldung

Mit der bereits registrierten E-Mail-Adresse und dem Passwort kann sich der Benutzer bei Tennis Companion anmelden. Wenn die E-Mail-Adresse und das Passwort in der Anfrage korrekt sind, werden ein Zugangs- und ein Aktualisierungstoken als Antwort vom Backend empfangen. Das Zugriffstoken wird vorübergehend während der eingeloggten Sitzung verwendet und das RefreshToken, sobald das Zugriffstoken abläuft. Das Zugriffstoken wird dann an die Blacklist gesendet und das RefreshToken verwendet, damit der Benutzer sich nicht jedes Mal neu anmelden und autorisieren muss.

Wenn sich der Benutzer abmeldet, geschieht zweierlei: Erstens werden die Token an die Blacklist geschickt und der Zugriff wird vollständig unterbunden, und zweitens wird eine Nachricht an RabbitMQ gesendet, der zu entnehmen ist, dass sich der Benutzer gerade abgemeldet hat. Somit weiß der Review-Service, dass der Zugriff auf die Bewertungen ebenfalls unterbunden werden muss.

EPIC 3: Benutzerkategorien und Suche

Die Benutzer werden hier nach drei Kategorien abgefragt: beliebte Benutzer, neueste Benutzer und empfohlene Benutzer. Die empfohlenen Benutzer werden nur angemeldeten Benutzern angezeigt, während die beliebten und neuesten für jeden Besucher des Portals verfügbar sind. Gleichzeitig wird eine Suche nach einem bestimmten Benutzer über das Suchfeld in ReactJS möglich sein, mit dem die Benutzer entsprechend gefiltert werden. Außerdem wird es zwei zusätzliche und unterschiedliche Endpunkte geben, um die Benutzer nach ihrer Verfügbarkeit oder nach ihrer ID filtern zu können.

EPIC 4: Profilverwaltung

Hier werden die meisten CRUD-Funktionen (Create, Read, Update/Patch, und Delete) erstellt. Diese werden von den APIs des User-Services abgeleitet, wie in den Tabellen 4.1 und 4.2 zu sehen ist. Das bedeutet, dass der Benutzer sein Profil entsprechend aufrufen, aktualisieren und auf Wunsch löschen kann. Der Administrator vermag alle Benutzer aufzurufen und bei Bedarf zu löschen.

Nr.	CRUD	Endpoint	Permission	Task	EPIC
1	GET	/reviews/api/users-ratings/<str:username>/	isAAuthz	gibt alle Benutzernamen hinterlassenen Bewertungen zurück.	Review Management
2	GET	reviews/api/ratings/<str:pk>	isAuth	Rückgabe einer spezifischen Review-Details	Review-Management
3	GET	reviews/api/ratings/my-reviews/<str:username>/	isAuth	die eigene Bewertungen zurückgeben	Review-Management
4	POST	reviews/api/rate/<str:rater_username>/	isAuth	Erstellung einer Bewertung zu einem Bewerter	Review Management
5	POST	reviews/api/reply-to-review/<str:pk>/<str:username>/<str:o_username>/	isAuth	Beantwortung einer Review-ID mit dem Beantworter und einem bestimmten Review-Rezendenten	Review-Management
6	UPDATE	reviews/api/update-review/<str:pk>/<str:username>/	isAuth	eine spezifische Bewertungs-ID aktualisieren	Review Management -
7	DELETE	reviews/api/delete-review/<str:pk>/<str:username>/	isAAuthz	Löschen der BewertungsID eines bestimmten Benutzernamen	Review-Management

Tabelle 4.2: Initiative 2 - Review-Management API's

Initiative 2: Review-Management

Im Review-Management-Service gibt es nur ein EPIC, das die CRUD-Funktionen dieses Services darstellt. Einige der Funktionen müssen den Benutzernamen und die User-ID des User-Services übernehmen, um ihr Ziel erreichen zu können. Diese Funktionen, die in isolierte und nicht isolierte Aufgaben aufgeteilt wurden, sind in Tabelle 4.3 und Tabelle 4.4 dargestellt.

Dies kurze Beschreibung des EPICs lautet wie folgt:

EPIC 1: CRUD von Bewertungen

Da die URLs und der Rumpf der Anfragen des Review-Services angepasst werden, um spezifische und korrekte Daten vom User-Service zu erhalten, werden deren Aufgaben geklärt.

In der ersten URL ist zu erkennen, dass der Benutzername des Rezensenten am Ende der URL zu finden ist. Dies ermöglicht es dem Administrator, alle Bewertungen dieses Benutzers aufzurufen, die von der URL geholt werden, sobald das Benutzerprofil angeklickt wird.

```
http://localhost:8080/reviews/api/users-ratings/<str:username>/
```

Bei der zweiten URL wird nur die ID der Bewertung benötigt, um die richtige Bewertung zu finden.

```
http://localhost:8080/reviews/api/ratings/<str:pk>
```

```
http://localhost:8080/reviews/api/ratings/my-reviews/<str:username>/
```

Bei der vierten URL wird nur der bewertete Benutzer im Hauptteil der Anfrage benötigt und in der URL wird der aktuell angemeldete Benutzer aus der URL geholt.

```
http://localhost:8080/reviews/api/rate/<str:rater_username>/
```

Diese URL ist ein wenig komplizierter und es ist erkennbar, dass es drei URL-Parameter gibt. Der erste dient dazu, die Bewertungs-ID (Review-ID) zu erhalten. Der zweite, der Benutzername, ist der Benutzer, der versucht, auf diese Bewertung zu antworten. Das dritte Argument ist der Benutzername des Bewerters, der abgerufen wird, sobald der bewertete Benutzer auf die Bewertung klickt, um den Benutzernamen des Bewerters aus der URL zu erhalten. Das bedeutet, dass der eingeloggte Benutzer auf eine Bewertung mit einer bestimmten Bewertungs-ID antwortet, die mit dem Benutzernamen eines bestimmten Bewerters verbunden ist.

```
http://localhost:8080/reviews/api/reply-to-review/<str:pk>/<str:username>/<str:o_username>/
```

Auch bei dieser URL gibt es drei bedeutende Faktoren. In den URL-Parametern werden die Review-ID und der Benutzername des Bewerters der URL entnommen. Der Benutzername des bewerteten Benutzers wird jedoch mit Hilfe der Review-ID abgerufen und die Antwort, die den Benutzernamen des bewerteten Benutzers enthält, wird verwendet.

```
http://localhost:8080/reviews/api/update-review/<str:pk>/<str:username>/
```

In der letzten URL sind nur die Review-ID und der Benutzername des angemeldeten Benutzers (admin) von Bedeutung. Dadurch wird dem Review-Service mitgeteilt, dass ein Benutzer, der vom User-Service geholt wird, mit den nötigen Berechtigungen einen Review mit einer bestimmten Review-ID löschen möchte

```
http://localhost:8080/reviews/api/delete-review/<str:pk>/<str:username>/
```

4.7 RabbitMQ

In diesem Unterabschnitt wird die Interaktion zwischen dem User-Service und dem Review-Service unter Verwendung von RabbitMQ beschrieben und anhand von Abbildung 4.1.1 eine Erläuterung dessen gegeben, was dort geschieht.

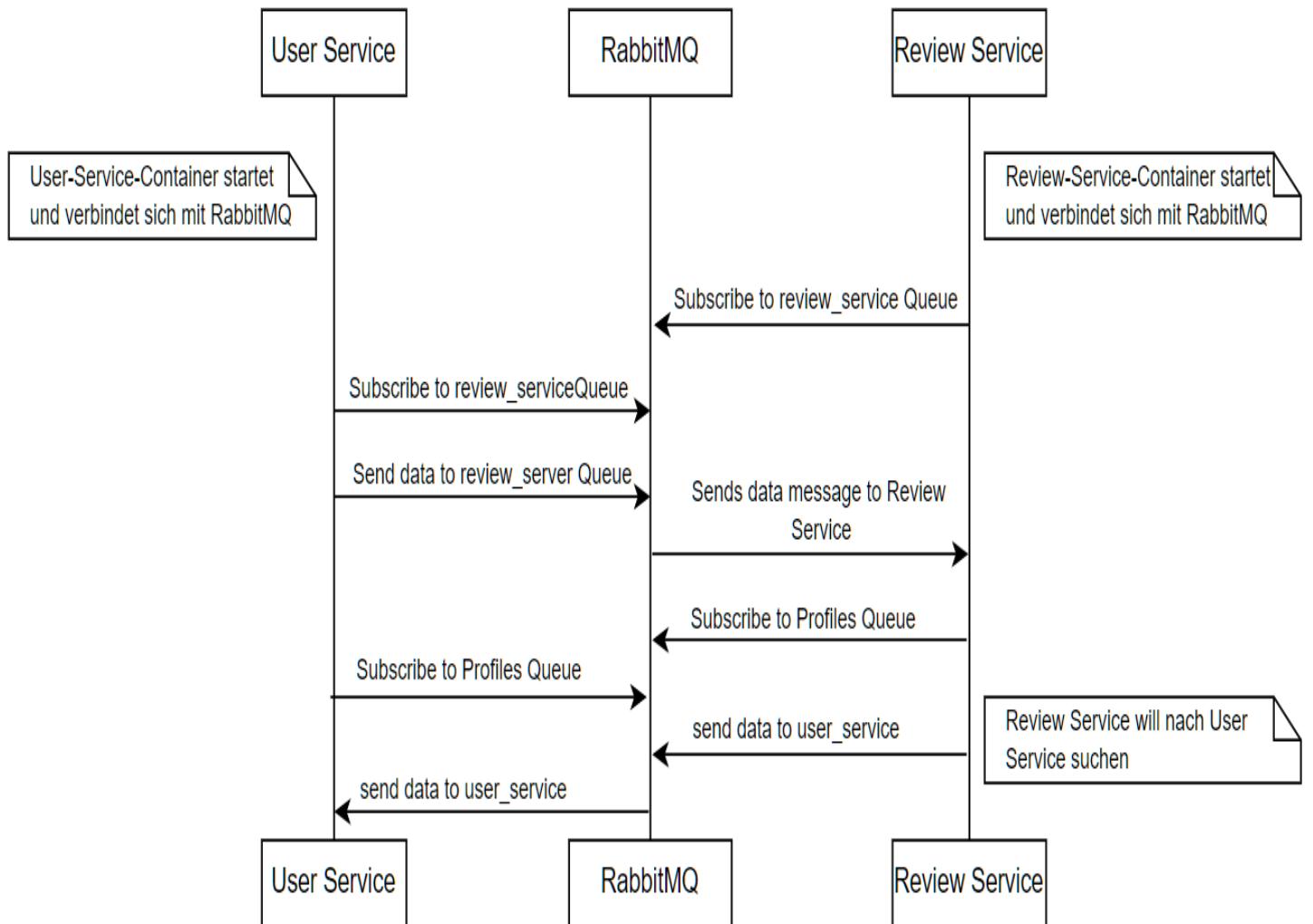


Abbildung 4.7.1: RabbitMQ - publish und subscribe für die Veröffentlichung und den Konsum von Nachrichten von dem RabbitMQ-Server (Quelle: eigene Darstellung)

In Abbildung 4.5.1 findet die interne Kommunikation zwischen den beiden Services ohne die Verwendung von RESTAPIs statt. Mit RabbitMQ abonniert der Review-Service zu der `review_service`-Queue und wartet auf jede Änderung, die eintritt. Das heißt, sobald der User-Service etwas im Message-Broker über die `review_service`-Queue veröffentlicht, wird der Review-Service darüber informiert, dass eine Nachricht oder Daten darauf warten, konsumiert zu werden. Sobald die Daten vom Review-Service aufgenommen werden, können

sie entsprechend verwendet werden, um bestimmte Aufgaben innerhalb dieses unabhängigen Services durchzuführen.

Gleichzeitig abonniert der User-Service die **Profiles**-Queue, so dass, falls der Review-Service etwas über diese Queue veröffentlicht oder eine Änderung stattfindet, die für den User-Service von Bedeutung ist, dieser entsprechend informiert wird, die Daten konsumiert und in seinem eigenen Service speichert.

Isolierte und nicht-isolierte Aufgaben

Die folgende Tabelle 4.4 verdeutlicht, welche Aufgaben aus den Anforderungstabellen 3.1 und 3.2. isoliert ohne den anderen Service erledigt werden können und für welche Aufgaben ein Datenaustausch zwischen den beiden Services notwendig ist. So kann festgestellt werden, welche Aufgaben innerhalb des Review-Services ohne den User-Service erledigt werden können und welche anderen Aufgaben Informationen vom User-Service benötigen. So lässt sich leichter ermitteln, welche Methoden und Aufrufe veröffentlicht werden sollten, wenn sie in einem der Services geändert werden.

Isolierte Aufgaben		
Nr.	User-Service	Review Service
1	get registered users	alle Bewertungen anzeigen
2	get popular users	
3	get latest users	
4	get recommended users	
5	get user details	
6	get my own profile	
7	filter available users	
8	filter users by id	
9	filter users by strength	
10	post new user Image	
11	update description text	
12	update description text	

Tabelle 4.3: Isolierte Aufgaben ohne Kommunikationsbedarf zwischen den Services

Nicht-Isolierte Aufgaben			
Nr.	publish	User-Service	consume
1	username, ID & Zugriffsrechte	Benutzerregisterierung über eine private EMail-Adresse	Anzahl an Rezensionen
2	username & ID & Zugriffsrechte	Benutzerregisterierung über ein Google-Mail-Konto	
3	Anmeldungsstatus & username	Benutzeranmeldung über eine private Email-Adresse	
4	Anmeldungsstatus & username	Benutzeranmeldung über ein Google-Mail-Konto	
5	Anmeldungsstatus & username	Benutzerabmeldung	
6	username & ID	Lösung eines Benutzers	

Tabelle 4.4: User-Service-Tasks, die über den RabbitMQ-Server veröffentlicht und konsumiert werden

In Tabelle 4.4 gibt es fünf verschiedene Situationen, die wie folgt erklärt werden:

1. Wenn ein Benutzer im User-Service erstellt wird, veröffentlicht der Service den Benutzernamen, die Zugriffsrechte (ob der Benutzer ein Admin ist) und die ID dieses registrierten Benutzers an RabbitMQ. Der Review-Service hört RabbitMQ ab und konsumiert diese Informationen, sobald sie dort ankommen, sodass er sie nutzen kann, um einem bestimmten Benutzernamen innerhalb des Review-Services eine Review-ID zuzuweisen, ohne alle Informationen über den Benutzer kennen zu müssen.
2. Wenn sich ein Benutzer mit seinem Google-Mail-Konto anmeldet, läuft derselbe Prozess ab, wie unter Nummer 1. beschrieben, und nur der Benutzername und die ID werden weitergegeben.
3. In der dritten und vierten Situation müssen dem Review-Service Informationen über den Anmeldungsstatus des Benutzers mitgeteilt werden, das heißt, ob er eingeloggt ist oder nicht. So kann der Review-Service feststellen, ob der Benutzer bereits angemeldet und authentifiziert ist oder nicht. Wenn dies der Fall ist, kann der Benutzer Aufrufe innerhalb des Review-Services tätigen, die eine Authentifizierung und Anmeldung erfordern.
4. In der fünften Situation wird wiederum der Review-Service informiert, wenn sich der Benutzer abgemeldet hat und keine Authentifizierung mehr vorliegt, um bestimmte autorisierte Aufgaben innerhalb des Review-Services durchzuführen.
5. In der sechsten und letzten Situation, beim Löschen eines Benutzers, wird der Review-Service darüber informiert, alle Links zu ihm aus seiner Datenbank zu entfernen.

Nicht-Isolierte Aufgaben			
Nr.	consume	Review-Service	publish
1	Anmeldungsstatus & username	Bewertung hinzufügen	Anzahl an Rezensionen
2	username, ID & Zugriffsrechte	Bewertung löschen	aktualisiertes Rating eines Benutzers
3		alle Bewertungen der Profil-ID anzeigen	
4		auf eine Bewertung antworten	
5		eine Bewertung bearbeiten	
6		eine Bewertung löschen	

Tabelle 4.5: Review-Service-Tasks, die über den RabbitMQ-Server veröffentlicht und konsumiert werden.

In Tabelle 4.5 werden nur zwei verschiedene Nachrichten veröffentlicht: eine, wenn eine Bewertung hinzugefügt wird, und eine weitere, wenn sie gelöscht wird.

Kapitel 5

Implementierung

In diesem Schritt wird der Programmcode entsprechend der Vorplanung, Methoden und Entwurfsmuster der vorherigen Schritte installiert. In dieser Phase wird mittels Internetrecherche neuer Programmcode implementiert und Erfahrungen von Dozenten und Studierenden werden bei Schwierigkeiten einbezogen. Jede Funktion und jedes Feature werden iterativ implementiert und den Anforderungen der Anwendung entsprechend ergänzt.

Jede Funktion und jedes Feature werden iterativ implementiert und den Anforderungen der Anwendung entsprechend ergänzt.

5.1 Systemarchitektur

Im folgenden Diagramm ist eine detaillierte Darstellung des Systems zu sehen. Im User-Service gibt es vier Container:

1. den Benutzerdienst-Container, der alle Informationen über die Anwendung enthält und
2. einen Container, der für die Datenbankverbindung zuständig ist. Wie im Diagramm zu sehen ist, wird PostgreSQL verwendet.
3. Der dritte Container ist für die Ausführung von RabbitMQ verantwortlich und hält die Verbindung aufrecht, wenn eine Nachricht gesendet oder empfangen wird.
4. Der letzte und vierte Container ist der NGINX, der als API-Gateway zwischen den drei Services dient.

Außerdem gibt es den Überprüfungsdiensst, der drei Container enthält, die jeweils in der Abbildung dargestellt sind. Der dritte Dienst ist der Frontend-Service, der nur einen Container für die Client-App besitzt. Wie im Diagramm zu sehen ist, teilt NGINX Informationen über Medien und statische Dateien mit dem User-Service-Container. Außerdem kann NGINX auch die beiden anderen Services über ein externes Netzwerk ermitteln, das in diesen Services eingerichtet ist.

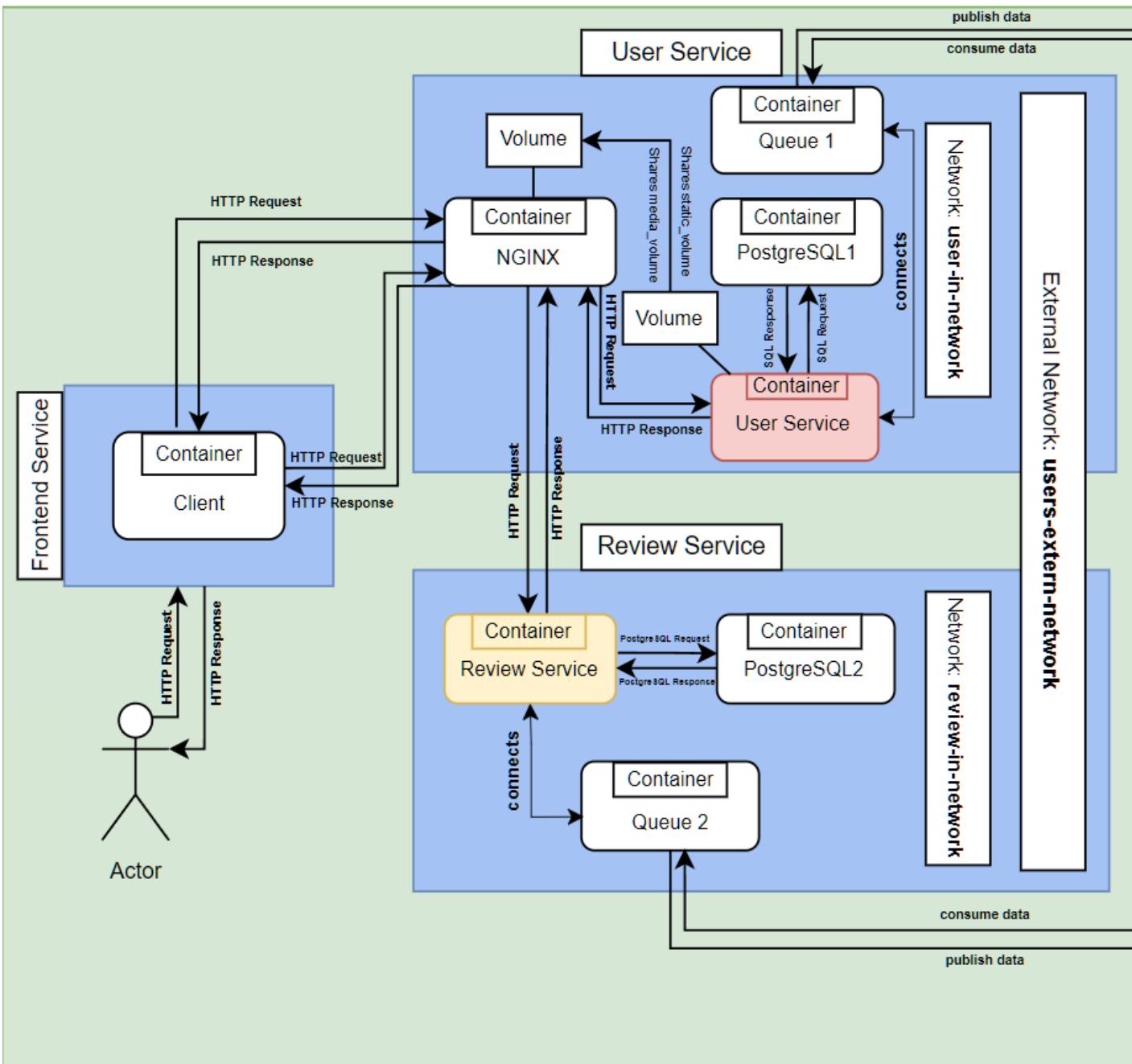


Abbildung 5.1.1: Die Systemarchitektur zeigt die Verbindung zwischen den verschiedenen Containern der Services und was in jedem davon passiert (Quelle: eigene Darstellung)

5.2 Technologien

In diesem Abschnitt wird die Konfiguration von Django, Docker, NGINX, RabbitMQ und ReactJS mit Hilfe von Abbildungen demonstriert.

5.2.1 Django

Django hat, wie bereits erwähnt, eine für die Konfiguration bedeutende Datei, die `settings.py`-Datei. Diese wurde aus Gründen der Organisation und Struktur in drei verschiedene Dateien unterteilt. Die Hauptkonfiguration befindet sich unter dem Ordner „`settings`“ in einer Datei namens „`base.py`“. Die Verbindungskonfiguration zur Datenbank sowie zu den E-Mail-Hosts ist in der Datei „`development.py`“ abgelegt. Im Folgenden sind zwei Bilder mit der Konfiguration zu sehen, die für das Funktionieren der Anwendung notwendig ist.

Dateipfad: `user_management/settings/base.py`

```
27 DJANGO_APPS = [
28     'django.contrib.admin',
29     'django.contrib.auth',
30     'django.contrib.contenttypes',
31     'django.contrib.sessions',
32     'django.contrib.messages',
33     'django.contrib.staticfiles',
34     'django.contrib.sites',
35 ]
36
37 SITE_ID = 1
38
39 THIRD_PARTY_APPS = [
40     "rest_framework",
41     "rest_framework_simplejwt",
42     'rest_framework_simplejwt.token_blacklist',
43     'corsheaders',
44     'phonenumbers',
45     'django_countries',
46     'knox',
47     'rest_framework_swagger',
48     'django_filters'
49 ]
50
51 LOCAL_APPS = [
52     "apps.manage_user",
53     "apps.common",
54     'apps.profiles',
55     'apps.ratings',
56     'apps.email_requests',
57     'custom_user',
58 ]
59
60 INSTALLED_APPS = DJANGO_APPS + THIRD_PARTY_APPS + LOCAL_APPS
```

(a) Konfiguration und Verbindungen von internen und externen Apps

```
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(days=5),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=90),
    'ROTATE_REFRESH_TOKENS': True,
    'BLACKLIST_AFTER_ROTATION': True,
    'ALGORITHM': 'HS256',
    'SIGNING_KEY': env("SIGNING_KEY"),
    'VERIFYING_KEY': None,
    'AUTH_HEADER_TYPES': (
        "Bearer",
        "JWT",
    ),
    'USER_ID_FIELD': 'id',
    'USER_ID_CLAIM': 'user_id',
    'AUTH_HEADER_NAME': "HTTP_AUTHORIZATION",
    'AUTH_TOKEN_CLASSES': ('rest_framework_simplejwt.tokens.AccessToken',),
    'TOKEN_TYPE_CLAIM': 'token_type',
}
```

(b) Konfiguration von der Authentifizierung über JWT

Abbildung 5.2.1: Manche Info. in der Konfigurationsdateien von Django

Daneben gibt es eine Konfiguration zur Erstellung von Logs bei jedem Lauf der Anwendung mittels `logging.config`. Diese Logs werden unter einem Ordner namens „`logs`“ gespeichert.

5.2.2 Docker

Ein Beispiel innerhalb der Docker-Compose- und Dockerfile-Dateien wird hier gezeigt und erklärt.

Docker Compose

In Abbildung 5.2.2 (a) ist Folgendes zu beachten:

- der Speicherort des Dockerfiles und der Befehl ,/start‘, der eine Bash-Datei mit wichtigen Befehlen zum Ausführen eines Django-Projektes ausführt.
- Außerdem gibt es die env_file, die alle Werte von Variablen enthält, die in verschiedenen Dateien verwendet werden und die geheim bleiben müssen.
- Am Ende der Abbildung befinden sich auch die Volumes und Netzwerke. Die Volumes speichern die gesamte App innerhalb des Containers an einem Ort namens ,app‘, legen die Media- und Static-Ordner ebenfalls an diesem Ort ab und erstellen schließlich einen neuen Ordner namens ,imgs‘ innerhalb von ,mediafiles‘, um alle statischen Bilder zu speichern, die manuell ohne die Verwendung des Admin-Backends von Django gespeichert wurden.
- Das ,networks‘-Tag fügt den gesamten Container in ein Netzwerk namens ,tennis-react‘ ein, sodass andere Container innerhalb desselben Projekts den ,user_manage_app‘-Container finden können.

In Abbildung 5.2.2 (b) ist die Logik fast dieselbe wie in der vorherigen Abbildung, jedoch mit einer neuen Dockerfile, die die Konfiguration für NGIX enthält. Außerdem gibt es den Port 8080, über den alle Services geleitet werden.

Dateipfad: docker-compose.yml

```
services:  
  user_manage_api3:  
    build:  
      context: .  
      dockerfile: ./docker/local/django/Dockerfile  
    command: /start  
    restart: on-failure  
    image: user_api  
    container_name: userapp3  
    env_file:  
      - .env  
    volumes:  
      - ./app  
      - static_volume:/app/staticfiles  
      - media_volume:/app/mediafiles  
      - ./mediafiles/imgs:/app/mediafiles/imgs/  
    depends_on:  
      - postgres-db  
    networks:  
      - tennis-react
```

(a) Konfiguration von der User Service API

```
nginx3:  
  restart: always  
  depends_on:  
    - user_manage_api  
  container_name: nginx_container3  
  volumes:  
    - static_volume:/app/staticfiles/  
    - media_volume:/app/mediafiles/  
    - ./mediafiles/imgs:/app/mediafiles/imgs/  
  build:  
    context: ./docker/local/nginx  
    dockerfile: Dockerfile  
  ports:  
    - "8080:80"  
  networks:  
    - tennis-react # this network
```

(b) Konfiguration von NGINX

Abbildung 5.2.2: Die Konfigurationen von den user_manage_api3 und NGINX Services in der docker-compose.yml Datei

In Abbildung 5.2.3 (a) ist eine weitere Dockerfile zu sehen, die die Konfiguration von RabbitMQ enthält, wobei ein entscheidender Befehl (‘python consumer.py’) hinzugefügt wurde, der den Consumer in der Anwendung startet, um alle Nachrichten zu konsumieren.

In Abbildung 5.2.3 (b) ist zu erkennen, dass der Name des Netzwerks ‘users_network’ lautet. Damit wird ein externes Netz geschaffen, mit dem sich die anderen Services verbinden können. Das bedeutet, dass die verschiedenen Services voneinander wissen, wenn sie das Netzwerk zu den anderen Services hinzufügen, wie in der letzten Abbildung zu sehen ist.

Dateipfad: docker-compose.yml und review_service/docker-compose.yml

```
queue3:  
  build:  
    context: .  
    dockerfile: ./docker/local/rabbitmq/Dockerfile  
  image: user_queue_image  
  container_name: user_queue3  
  env_file:  
    - .env  
  command: 'python consumer.py'  
  restart: on-failure  
  depends_on:  
    - postgres-db  
  networks:  
    - tennis-react
```

(a) RabbitMQ Konfiguration

```
networks:  
  tennis-react:  
    name: users_network  
  
volumes:  
  postgres_data:  
  static_volume:  
  media_volume:
```

(b) Netzwerkkonfiguration von dem User-Service

```
networks:  
  app-tier:  
    driver: bridge  
  my-tennis-react:  
    external:  
      name: users_network
```

(c) Netzwerkkonfiguration von dem Review-Service

Abbildung 5.2.3: Die Konfigurationen vom RabbitMQ-Service in der Konfiguration von den internen und externen Netzwerken in der docker-compose.yml Datei

Die Konfiguration des restlichen Review-Services wird nicht gezeigt, da sie der des User-Services ähnlich ist.

Dockerfile

In diesem Unterabschnitt wird eine Demonstration des Inhalts der einzelnen Dockerfiles vorgenommen.

In der nachstehenden Abbildung wird zunächst ein Verzeichnis erstellt, in dem die Anwendung innerhalb des Containers platziert werden soll. Anschließend werden zwei Verzeichnisse angefertigt, die die Media- und Staticfiles enthalten. Dann werden alle notwendigen Updates vorgenommen, damit die Datenbank und andere Dinge problemlos ausgeführt werden können.

Innerhalb der Anwendung gibt es die Datei „requirements.txt“, die alle externen Bibliotheken enthält, die mit Hilfe des „pip“-Paketverwaltungsprogramms installiert werden sollen. Sie wird in den Container kopiert, so dass alle notwendigen Pakete innerhalb des Containers installiert sind, bevor die Anwendung startet.

Am Ende der Datei werden zwei weitere wichtige Bash-Dateien kopiert und mit Ausführungsrechten versehen. Dies sind die Dateien „entrypoint“ und „start“. Erstere enthält eine Konfiguration für die Verbindung zur Postgresql-Datenbank im Container und gibt im Fehlerfall eine Meldung über Probleme bei der Verbindung zur Datenbank aus. Auf der anderen Seite enthält die „start“-Datei alle nötigen Befehle, um die Django-Anwendung zu starten und alle Migrationen zur Datenbank durchzuführen, bevor der Server startet.

Dateipfad: docker/local/django/Dockerfile

```
FROM python:3.10.0-slim-buster

ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

ENV APP_HOME=/app
RUN mkdir $APP_HOME
RUN mkdir $APP_HOME/staticfiles
RUN mkdir $APP_HOME/mediafiles

WORKDIR $APP_HOME

RUN apt-get update \
    && apt-get install -y build-essential \
    && apt-get install -y libpq-dev \
    && apt-get install -y gettext \
    && apt-get -y install netcat gcc postgresql \
    && apt-get purge -y --auto-remove -o APT::AutoRemove::RecommendsImportant=false \
    && rm -rf /var/lib/apt/lists/*

RUN python -m pip install --upgrade pip
RUN #pip uninstall psycopg2

COPY ./requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt

COPY ./docker/local/django/entrypoint /entrypoint
RUN sed -i 's/\r$/\g' /entrypoint
RUN chmod +x /entrypoint

COPY ./docker/local/django/start /start
RUN sed -i 's/\r$/\g' /start
RUN chmod +x /start

ENTRYPOINT [ "/entrypoint" ]
```

Abbildung 5.2.4: Dockerfile von dem user_manage_api3-Container des User-Services

In Abbildung 5.2.5 (a) ist die Konfigurations-Dockerfile für den RabbitMQ-Container zu sehen. Sie verschiebt die gesamte Anwendung in ihren Container, kopiert und installiert die erforderlichen Pakete und führt die Datei „consume.py“ mit Hilfe des „command“-Tags in der Datei „docker-compose“ aus.

In Abbildung 5.2.5 (b) wird ein Image von NGINX verwendet. Anschließend wird bei Verfügbarkeit zunächst die bereits vorhandene Konfigurationsdatei „default.conf“ für NGINX, die sich innerhalb des Containers befindet, entfernt. Daraufhin wird die neu erstellte Konfigurationsdatei vom lokalen Rechner in den Container kopiert, um die richtige Konfiguration für die verschiedenen Services einzustellen.

Dateipfad: docker/local/django/Dockerfile und docker/local/nginx/Dockerfile

```
FROM python:3.10.0-slim-buster

ENV PYTHONUNBUFFERED 1
ENV APP_HOME=/app
RUN mkdir $APP_HOME
RUN mkdir $APP_HOME/staticfiles

WORKDIR $APP_HOME

COPY ./requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt

COPY . /app
```

(a) Dockerfile von dem RabbitMQ-Container des User-Services

```
FROM nginx:1.21.3-alpine

RUN rm /etc/nginx/conf.d/default.conf

COPY ./default.conf /etc/nginx/conf.d/default.conf
#COPY ./COPY
```

(b) Dockerfile von dem NGINX-Container des User-Services

Abbildung 5.2.5: Dockerfiles von RabbitMQ und NGINX

5.2.3 RabbitMQ

In diesem Unterabschnitt wird die Konfiguration von RabbitMQ demonstriert. Es gibt zwei bedeutende Dateien, die erstellt wurden, um die Nachrichten von RabbitMQ zu veröffentlichen und zu konsumieren: „produce.py“ und „consume.py“. Der Publisher des User-Services und der Consumer des Review-Services werden nur verdeutlicht, da die Konfiguration der Gegenseite annähernd gleich ist.

producer.py

Die Zeilen 38 und 39 sind für den Aufbau einer Verbindung mit dem RabbitMQ-Server zuständig.

Dann wird in Zeile 42 eine Funktion ‚publish‘ erstellt, die das Senden der zu übermittelnden Nachricht behandelt.

Es werden auch ‚properties‘ als Inhalt der ‚method‘ und diese wiederum als einer der Parameter von ‚channel.basic.publish‘ übergeben.

Der Default-Exchange, der durch ‚exchange=‘ gekennzeichnet ist, wird verwendet und ermöglicht es, die Warteschlange (queue) anzugeben, die die Nachricht durchlaufen soll.

Die Warteschlange wurde auf ‚review_service‘ eingestellt und muss in der Anwendung, die die Nachricht erhält, deklariert werden. Anhand der in den Properties eingegebenen Informationen oder Zeichenketten kann der Consumer erkennen, welche Nachricht veröffentlicht wurde. Der Code in Abbildung 5.2.7 zeigt beispielhaft, wie eine Nachricht veröffentlicht wird, wenn sich ein Benutzer im Tennis-Companion-Portal registriert.

Dateipfad: apps/profiles/producer.py

```
34  class RabbitMq():
35
36      def __init__(self):
37
38          self._connection = pika.BlockingConnection(pika.URLParameters(env("RABBITMQ_HOST")))
39          self._channel = self._connection.channel()
40          self._channel.queue_declare(queue="review_service")
41
42      def publish(self, method, body):
43          properties = pika.BasicProperties(method)
44          self._channel.basic_publish(exchange="",
45                                      routing_key="review_service",
46                                      body=json.dumps(body), properties=properties)
47
48          print("Published Message: {}".format(body), "--> ", properties.content_type)
49          self._connection.close()
```

Abbildung 5.2.6: Die Verbindungsconfiguration vom Producer zu RabbitMQ im User-Service

Dateipfad: apps/profiles/signals.py

```
23     if created:  
24         Profile.objects.create(user=instance)  
25         instance.profile.save()  
26  
27         profile_id_str = str(instance.profile)  
28         data = {'id': profile_id_str,  
29                 "username": instance.username,  
30                 'is_admin': str(instance.is_staff)}  
31  
32         p = RabbitMq()  
33         RabbitMq.publish(p, 'profile_created', data)  
34  
35         logger.info(f"{instance}'s profile created {dict}")
```

Abbildung 5.2.7: Die Veröffentlichung einer Nachricht bei Registrierung eines Benutzers im User-Service

Von Zeile 28 bis 30 sind die Daten zu sehen, die auf dem RabbitMQ-Server veröffentlicht werden sollen. In Zeile 33 wird im ersten Argument RabbitMQ instanziert, im zweiten wird die Property „profile_created“ eingegeben und im dritten sind die zu sendenden Daten enthalten. Der Property-Name im Consumer muss ebenfalls „profile_created“ heißen, damit klar ist, um welche Nachricht es sich handelt.

Die Veröffentlichung anderer Nachrichten funktioniert nach demselben Prinzip, das in Abbildung 5.2.7 dargestellt ist. Dabei müssen die zu veröffentlichten Daten entsprechend angepasst werden.

consumer.py

Da zunächst ein Zugriff auf ein Modell außerhalb des ‚review_service‘ erforderlich ist, muss das ‚DJANGO_SETTINGS_MODULE‘ wie in Abbildung 5.2.8 eingerichtet werden. eingerichtet werden.

Dateipfad: review_service/consumer.py

```
9
10    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "review_service.settings.development")
11    django.setup()
12
```

Abbildung 5.2.8: Die Einrichtung von ‚DJANGO_SETTINGS_MODULE‘ in consumer.py des Review-Services

Wie beim Producer wird in den Zeilen 48 und 49 eine Verbindung zum RabbitMQ-Server hergestellt. Die Warteschlange, die die Nachrichten empfängt, muss hier ebenfalls deklariert werden. Sie sollte den Namen haben, der im Parameter ‚routing_key‘ in der Funktion ‚channels.basic_publish‘ in der Datei ‚producer.py‘ im User-Service-Projekt deklariert wurde. In Zeile 50 wird RabbitMQ angewiesen, der Callback-Funktion in Abbildung 5.2.9 zu erlauben, Nachrichten aus der Review-Service-Warteschlange zu empfangen.

Dateipfad: review_service/consumer.py

```
29  class RabbitMqServerConfigure(metaclass=MetaClass):
30
31      def __init__(self, host=env("RABBITMQ_HOST"),
32                   queue='review_service'):
33
34          """ Server initialization """
35
36          self.host = host
37          self.queue = queue
38
39
40  class rabbitmqServer():
41
42      def __init__(self, server):
43
44          """
45          :param server: Object of class RabbitMqServerConfigure
46          """
47          self.server = server
48          self._connection = pika.BlockingConnection(pika.URLParameters(self.server.host))
49          self._channel = self._connection.channel()
50          self._tem = self._channel.queue_declare(queue=self.server.queue)
51          print("Server started waiting for Messages ")
```

Abbildung 5.2.9: Die Verbindungskonfiguration vom Producer zu RabbitMQ im Review-Service

In Abbildung 5.2.10 wird eine Callback-Funktion erstellt, die immer dann aufgerufen wird, wenn eine Nachricht empfangen wird. Dabei ist ‚ch‘ der Kanal, über den die Kommunikation erfolgt. Weiterhin ist ‚method‘ die Information über die Zustellung der Nachricht. ‚properties‘ sind benutzerdefinierte Eigenschaften der Nachricht und ‚body‘ ist die empfangene Nachricht.

In diesem Fall ist die Callback-Funktion für das Anlegen und Löschen von normalen und Admin-Benutzern zuständig. Sie teilt auch den Anmeldungs- und Abmeldungsstatus der Benutzer aus dem User-Service mit.

Dateipfad: review_service/consumer.py

```

54     @staticmethod
55     def callback(ch, method, properties, body):
56         print('Received in review_service')
57         data = json.loads(body)
58         print(data)
59
60         if properties.content_type == 'profile_created':
61             print("Information about the id and username: Id:", data['id'], " username: ", data['username'])
62             user_exists = Rater.objects.filter(username=data['username']).exists()
63
64             if user_exists == False:
65                 rater = Rater.objects \
66                     .create(username=data['username'])
67
68                 if data['is_admin'] == 'True':
69                     rater.is_admin = True
70                     rater.save()
71                     print('Rater has been created and is an admin user')
72                     rater.save()
73                     print("Rater and rated user Profiles have been created")
74                 else:
75                     print("User already saved")
76
77             if properties.content_type == 'user_signed':
78                 print("User ", data['username'], " just signed in")
79                 rater = Rater.objects.get(username=data['username'])
80
81                 if data['logged_status'] == "False":
82                     rater.is_signed = False
83                     rater.save()
84                     print("Rater just signed out")

```

Abbildung 5.2.10: Der Empfang verschiedener Nachrichten innerhalb der Callback-Funktion im Review-Service

Am Ende wird dem Server mitgeteilt, dass er mit dem Konsumieren des zuvor angegebenen Channels starten soll.

Dateipfad: review_service/consumer.py

```
96     def startserver(self):
97         self._channel.basic_consume(
98             queue=self.server.queue,
99             on_message_callback=rabbitmqServer.callback,
100            auto_ack=True)
101
102
```

Abbildung 5.2.11: Konfiguration, um Nachrichten von RabbitMQ im Review-Service zu konsumieren

5.2.4 NGINX

Die Angabe „stream“ definiert eine Gruppe von Servern, die an verschiedenen Ports lauschen können, falls es mehr als einen Server innerhalb desselben Services gibt. Sie tauschen sich dann mit Hilfe eines Load-Balancers untereinander aus. In diesem Fall ist zu erkennen, dass es von Zeile 1 bis 3 nur einen Server gibt, der über den Port 8004 lauscht. Dabei ist „userapp3“ der Name des Containers, der den Service „user_manage_api“ in der Docker-Compose-Datei „User service“ enthält. Aus den Zeilen 4 bis 10 ist ersichtlich, dass sowohl für den Review-Service als auch für den Frontend-Service dasselbe getan wurde. Defines a group of servers. Servers can listen on different ports. In addition, servers listening on TCP and UNIX-domain sockets can be mixed.

Dateipfad: user_management/docker/local/nginx/default.conf

```
1 upstream user_api {
2     server userapp3:8004;
3 }
4 upstream reviews_api {
5     server reviewapp:8003;
6 }
7
8 upstream frontend_api {
9     server frontend_api:3000;
10    resolver 127.0.0.11;
11 }
```

Abbildung 5.2.12: Das Hinzufügen von verschiedenen Servern oder Domänen in der NGINX-Konfigurationsdatei

Im Rest der Datei „default.conf“ werden die Einstellungen für die Header, den Port und die Pfade der Services definiert. Der NGINX-Server hört den Port 80 ab, wie Zeile 16 zeigt, und gibt den Port 8080 frei, der in der Datei „docker-compose.yml“ in Abbildung 5.2.2 (b) konfiguriert ist. Da es Anfragen zwischen verschiedenen Origins gibt, wird, wie in Zeile 17 gezeigt, ein Header hinzugefügt, der Cross-Origin-Requests erlaubt. „Cross-Origin Resource Sharing (CORS) W3C 2022 ist ein Mechanismus, der Webbrowsersn oder auch anderen Webclients Cross-Origin-Requests ermöglicht“. Der Stern zeigt an, dass alle Origins erlaubt sind. Die Konfiguration kann auch so eingestellt werden, dass sie nur bestimmte Origins akzeptiert wie „`http://review_api/..` usw. `always`“ behandelt auch die Fehlerantwort, falls das Backend mit Fehlern antwortet.

In Zeile 18 wird die Kopfzeile „Access-Control-Credentials“ auf „true“ gesetzt. Laut einem MDN-Artikel MDN 2022a teilt der Access-Control-Allow-Credentials-Response-Header den Browsern mit, ob sie die Antwort für den JavaScript-Code des Frontends freigeben sollen.

Wenn der Credentials-Modus einer Anfrage („Request.credentials“) „Include“ ist, geben die Browser die Antwort nur dann an den JavaScript-Code des Frontends weiter, wenn der Wert der Kopfzeile „Access-Control-Credentials“ „true“ ist.

Laut einem anderen MDN-Artikel MDN 2022b wird der Header „Access-Control-Allow-Headers“ als Antwort auf eine Preflight-Anfrage verwendet, die den Access-Control-Request-Header enthält, um anzugeben, welche HTTP-Header während der eigentlichen Anfrage verwendet werden können. Dies ist in Zeile 19 mit einem Stern (*) verzeichnet. In Zeile 20 werden die zulässigen Methoden für jede Anfrage festgelegt, wobei alle CRUD-Methoden entsprechend akzeptiert werden.

Unter „location“ ab Zeile 23 werden, wie in Abbildung 5.2.14 zu sehen ist, die Routen zu jeder Anwendung festgelegt und entsprechend zugelassen. Das Routing zum User-Service beginnt immer mit dem Präfix „/users“ und zum Review-Service mit „/reviews“, wie Zeilen 23 und 29 in Abbildung 5.2.14 zeigen. Das Gleiche gilt für die Admin-Panels der einzelnen Services sowie für das Frontend, das mit der Default-Location „/“ geroutet wird.

Dateipfad: user_management/docker/local/nginx/default.conf Dateipfad: user_management/docker/local/nginx/default.conf

```
13 server {
14
15     client_max_body_size 20M;
16
17     listen 80;
18     add_header Access-Control-Allow-Origin '*' always;
19     add_header Access-Control-Allow-Credentials 'true';
20     add_header 'Access-Control-Allow-Headers' '*' always;
21     add_header 'Access-Control-Allow-Methods' 'GET, POST, PUT, DELETE, OPTIONS' always;
22
23     location /users {
24         proxy_pass http://user_api;
25         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
26         proxy_set_header Host $host;
27         proxy_redirect off;
28     }
```

Abbildung 5.2.13: Die Konfiguration von Header und Hauptpfad zum User-Service

```
29     location /reviews {
30         proxy_pass http://reviews_api;
31         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
32         proxy_set_header Host $host;
33         proxy_redirect off;
34     }
35
36     location /admin {
37         proxy_pass http://reviews_api;
38         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
39         proxy_set_header Host $host;
40         proxy_redirect off;
41     }
42
43     location /superuser {
44         proxy_pass http://user_api;
45         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
46         proxy_set_header Host $host;
47         proxy_redirect off;
48     }
```

Abbildung 5.2.14: Die Hauptpfade zum Review-Service und zu den Administrationspannels der beiden Services

In der letzten Abbildung dieses Unterabschnitts ist die Location für die Media- und Static-Dateien angegeben.

Dateipfad: user_management/docker/local/nginx/default.conf

```
50          location /staticfiles/ {  
51              alias /app/staticfiles/;  
52          }  
53  
54          location /mediafiles/ {  
55              alias /app/mediafiles/;  
56          }
```

Abbildung 5.2.15: Die Pfade zu mediafiles und staticfiles

5.3 Model-Layer

Am Anfang der Model-Layer steht ein Entity-Relationship-Diagramm, das die Beziehungen zwischen den erstellten Objekten bzw. Modellen und allen darin verwendeten Attributen darstellt.

In den Abbildungen 5.3.1 und 5.3.2 zeigt der Punkt an jeder Linie die Art der Beziehung zwischen den Modellen an. Befindet sich nur ein Punkt am Ende einer Zeile, so handelt es sich um eine Eins-zu-Viele-Beziehung. Steht an beiden Seiten der Linie ein Punkt, wird eine Viele-zu-Viele-Beziehung angezeigt.

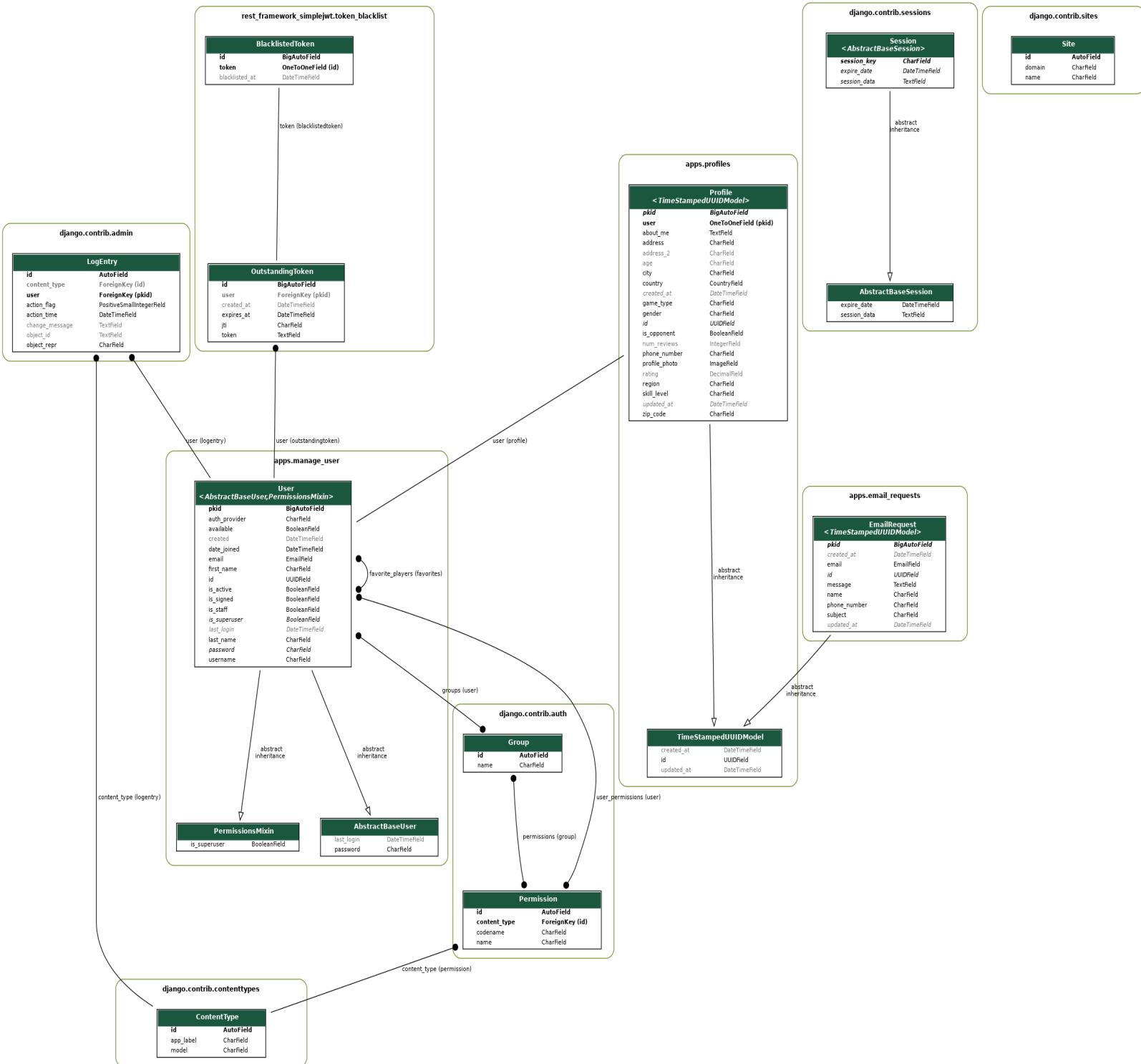


Abbildung 5.3.1: Entity-Relationship-Diagramm des User-Service

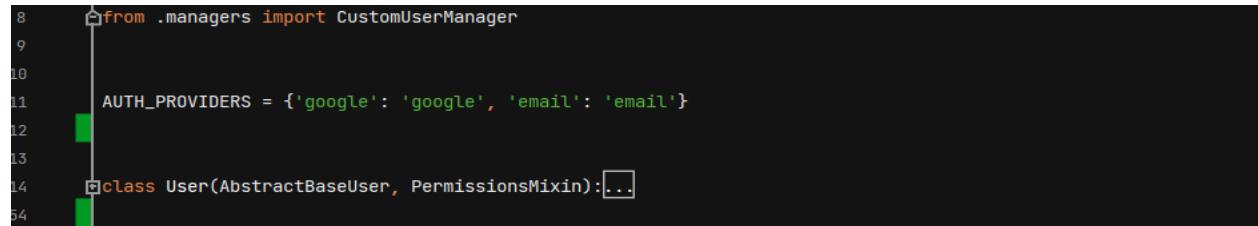
Initiative 1: User-Management

Zunächst wurde der Default-AbstractBaseUser, der mit Django ausgeliefert wird, erweitert und kann nun angepasst werden. Die Anpassung hilft bei der Verwendung einiger wesentlicher Attribute, die das Benutzermodell hat. AbstractBaseUser enthält nur die Authentifizierungsfunktionalität, aber keine eigentlichen Felder: Diese müssen beim Unterklassifizieren eingegeben werden. So können wir auch mit Attributen arbeiten wie:

- **is_staff**: Mit diesem von Django vorgegebenen Attribut kann ein Benutzer so eingestellt werden, dass er über Admin- oder Nicht-Admin-Berechtigungen verfügt. In der erstellten Klasse in Abbildung 5.3.2 wurde der Wert von `is_staff` auf `'False'` geändert, so dass alle neu registrierten Benutzer über Nicht-Admin-Zugriffsberechtigungen verfügen.
- **is_active**: Mit diesem anderen Attribut kann der Aktivierungsstatus der Benutzerkonten in aktiv oder nicht aktiv geändert werden, indem der Wert entweder auf True oder False gesetzt wird.

Außerdem werden zusätzliche Attribute wie Benutzername, E-Mail, Passwort usw. hinzugefügt. Der Benutzername ist eindeutig festgelegt, damit ein bestimmter Benutzername nicht doppelt vorkommt. Dies wird es später ermöglichen, verschiedene Benutzer innerhalb des Review-Services zu identifizieren, wobei der Benutzername ebenfalls eindeutig gesetzt wird und es möglich sein wird, herauszufinden, welcher Benutzername welcher Review-ID zugeordnet ist.

Dateipfad: user_management/apps/manage_user/models.py



```
8      from .managers import CustomUserManager
9
10
11     AUTH_PROVIDERS = {'google': 'google', 'email': 'email'}
12
13
14     class User(AbstractBaseUser, PermissionsMixin):...
```

The screenshot shows a code editor with a dark theme. It displays a Python file containing the definition of a User model. The code includes imports for managers and permissions, a list of authentication providers, and the User model itself which inherits from AbstractBaseUser and PermissionsMixin. Line numbers are visible on the left side of the code.

Abbildung 5.3.2: Der User-Model

Außerdem wird der BaseUserAdmin erweitert, da der Benutzer im Benutzermodell angepasst wurde; und er wird so angepasst, dass das Admin-Panel diejenigen Informationen anzeigt, die für den neu angepassten Benutzer im Benutzermodell aus Abbildung 5.3.2 benötigt werden.

Dateipfad: user_management/apps/manage_user/models.py



```
9 ① class UserAdmin(BaseUserAdmin):...
85
86
87 admin.site.register(User, UserAdmin)
```

The screenshot shows a code editor with a dark theme. It displays a single file named 'models.py'. The code defines a class 'UserAdmin' that inherits from 'BaseUserAdmin'. In the final line, there is a call to 'admin.site.register(User, UserAdmin)'. A yellow lightbulb icon is visible in the gutter between lines 86 and 87, indicating a potential issue or suggestion.

Abbildung 5.3.3: Die Konfiguration des User-Adminpanels

Darüber hinaus gibt es das Profilmodell, das eine Eins-zu-eins-Beziehung mit dem Benutzermodell hat. Das bedeutet, dass ein Benutzer nur ein Profil haben darf und dass ein Profil nur einem Benutzer gehören darf. Es erweitert auch das `TimeStampedUUIDModel`, wie in Abbildung 5.3.4 in Zeile 36 gezeigt. Der Grund dafür ist, dass „`uuid`“ verwendet wurde, um die IDs zu verschleiern, was es für Angreifer praktisch unmöglich macht, die IDs zu erraten. Das Problem mit „`uuid`“ ist jedoch, dass sie im großen Maßstab zu massiven Leistungsproblemen beim Einfügen führen und es keine schnelle „sort by id“-Chronologie gibt. Dieser Nachteil kann durch die Implementierung eines Pseudo-Primärschlüssels vermieden werden, und ebendiese Funktion erfüllt die konfigurierte `TimeStampedUUIDModel`, die von der App „common“ (Dateipfad: `apps/common/models.py`) innerhalb des Projekts erweitert wird. Außerdem werden Klassen erstellt, die eine Auswahlliste für die Optionen Gender, Skill-Level und Game-Type enthalten. Dies ermöglicht es dem Benutzer, eine Reihe von Optionen zwischen den gegebenen Informationen zu wählen. Am Ende der Abbildung befinden sich außerdem zwei Kommentare: „Blacklisted tokens“ und „Outstanding Tokens“. Dies sind die Tokens, die von einer Bibliothek eines Drittanbieters geerbt werden und bei der Erstellung von Zugangs- und Aktualisierungs-Tokens helfen sowie diese in der Datenbank speichern. Außerdem sind die Blacklisted Tokens nützlich, um aktive Zugangs-Tokens während einer aktiven Sitzung nach einer bestimmten Zeit zu blockieren und sie an eine Blacklist von Tokens zu senden, die keinen Zugang mehr haben. Mit dem Refresh-Token wird es dann möglich sein, einen kontinuierlichen Zugriff auf die aktive Sitzung zu haben. Die Refresh-Tokens werden auch rotieren, so dass dasselbe Refresh-Token nicht zweimal verwendet werden kann. Ein Refresh-Token, der zweimal verwendet werden kann, kann den Zugang anfällig für externe Hackerangriffe machen, was mit der Rotation jedoch nicht mehr möglich sein wird.

Dateipfad: user_management/apps/profiles/models.py

```
 9     User = get_user_model()
10
11
12     class Gender(models.TextChoices):...
13
14
15     class SkillLevel(models.TextChoices):...
16
17
18     class GameType(models.TextChoices):...
19
20
21
22     class Profile(TimeStampedUUIDModel):...
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
```

Abbildung 5.3.4: Der Profile-Model

Initiative: Review Management

Anhand eines Beispiels aus dem Buch „Designing Microservices with Django“ Hochrein 2019, S. 118, das in diesem Projekt als Analogie verwendet wird, wird erklärt, dass z. B. in der Situation des Projekts ‚Tennis Companion‘ ein harter Fremdschlüssel für das Feld ‚rater‘ mit dem Benutzerprofilmodell zu einer stärkeren Kopplung zwischen den Attributen führen kann. Aus diesem Grund sollten die Fremdschlüssele auf virtuelle Objekte verweisen, wobei das Fremdobjekt als Referenz betrachtet werden kann. Dadurch werden die Objekte weniger gekoppelt und vertrauensbasiert. Der Review-Service vertraut dem System, dass es einen Benutzer mit einem bestimmten Benutzernamen gibt, und kann in seiner eigenen Umgebung leben. Es gibt keine fest kodierten Datenbankregeln mehr, die Bewerter (Rater) aus dem Review-Service und die User-Profile aus den User-Services miteinander verbinden. Das bedeutet, dass es keine CASCADE-Löschtorgänge mehr gibt, was normalerweise bedeutet, dass die verknüpften Objekte manuell gelöscht werden müssen. Da jedoch RabbitMQ in diesem Projekt konfiguriert ist, wird RabbitMQ jedes Mal informiert, wenn ein Benutzer gelöscht wird, und der Bewerter im Review-Service wird automatisch ebenfalls gelöscht, sobald die geteilte Nachricht über diese Änderung beim Review-Service eintrifft.

In Abbildung 5.3.5 sind die Beziehungen zwischen den verschiedenen Modellen innerhalb des Review-Service und die Attribute der einzelnen Modelle dargestellt. Wie in der Abbildung zu sehen ist, verfügt der Rater über ein Attribut namens ‚username‘. Der verbrauchte Benutzername aus dem User-Service wird als Bewerter mit diesem Benutzernamen zur Datenbank des Review-Service hinzugefügt, ohne dass der Rest der Informationen über diesen spezifischen Benutzer bekannt sein muss, da er in seinem eigenen System verbleibt.

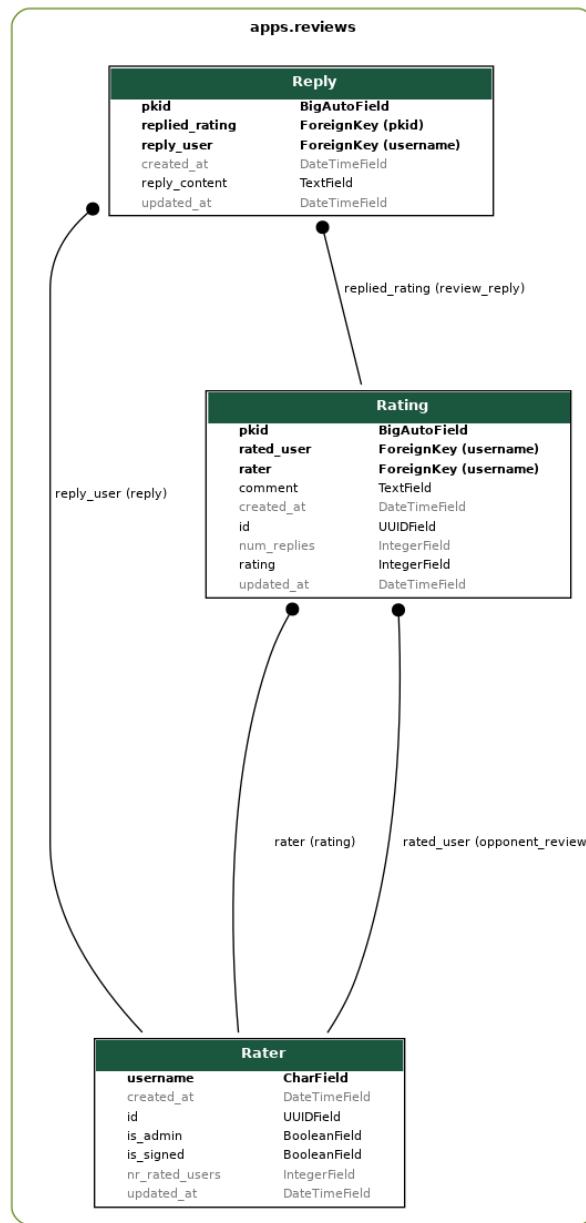


Abbildung 5.3.5: Entity-Relationship-Diagramm des Review-Service

5.4 Controller-Layer

Initiative 1: User-Management

In Ergänzung zu den Endpunkten, die in der Controller-Layer in Abschnitt 4.5 erwähnt wurden, werden hier einige der entsprechenden Funktionen beschrieben.

EPIC 1: Benutzerregisterierung

Während der Registrierung wird die Anfrage serialisiert und validiert. Der Validator ist eine Callable, die einen Wert annimmt und einen ValidationError auslöst, wenn bestimmte Kriterien nicht erfüllt sind. Außerdem benötigt er alle Attribute des Benutzers mit Ausnahme des Attributs `adresse_2`, das optional ist. All diese Daten werden über das Frontend gesendet, wo Fehler behandelt werden, falls ein Benutzer ein Feld vor dem Senden der Anfrage leer lässt.

Dateipfad: user_management/apps/manage_user/controller/user_register.py

```
8      # API to register to Tennis Companion
9      class RegisterAPI(CreateAPIView):...
```

Abbildung 5.4.1: Methode zur Registerierung eines Benutzers in das Portal

Außerdem ist in Abbildung 5.4.2 während der Registrierung zu sehen, dass in den Zeilen 26 bis 29 die Daten hinzugefügt werden, die an den Review-Service gesendet werden sollen. In Zeile 32 werden sie unter dem Property-Name `'profile_created'` veröffentlicht.

Dateipfad: apps/profiles/signals.py

```
20      @receiver(post_save, sender=AUTH_USER_MODEL)
21      def create_user_profile(sender, instance, created, **kwargs):
22          if created:
23              Profile.objects.create(user=instance)
24              instance.profile.save()
25
26          profile_id_str = str(instance.profile)
27          data = {'id': profile_id_str,
28                  "username": instance.username,
29                  'is_admin': str(instance.is_staff)}
30
31          p = RabbitMq()
32          RabbitMq.publish(p, 'profile_created', data)
```

Abbildung 5.4.2: Veröffentlichung des Benutzernamens während der Erstellung eines Benutzers im User-Service

EPIC 2: Benutzeranmeldung

In Abbildung 5.4.3 sind die Storys aus Tabelle 3.4 umgesetzt. In der ersten Klasse in Zeile 16 werden `username` und `logged_status` unter dem Property-Name `user_signed` mit dem Review-Service geteilt, sobald sich der Benutzer anmeldet, wie in Abbildung 5.4.4 gezeigt.

Dateipfad: user_management/apps/manage_user/controller/user_login.py

```
42     # API to login with a private email-address
43     class LoginAPI(APIView):...
49
50     # API to login with gmail account
51     class GoogleSocialAuthView(GenericAPIView):...
55
56     # API to blacklist a token
57     class BlacklistTokenUpdateView(APIView):...
121
122     ↳ class PasswordContextMixin:...
131
132     # API to reset password
133     class PasswordResetView(PasswordContextMixin, FormView):...
162
163     INTERNAL_RESET_SESSION_TOKEN = "_password_reset_token"
164
165     # API to show that password reset is done
166     class PasswordResetDoneView(PasswordContextMixin, TemplateView):...
170
171     # API to confirm the password reset
172     class PasswordResetConfirmView(PasswordContextMixin, FormView):...
256
257     # API to show the password reset completion
258     class PasswordResetCompleteView(PasswordContextMixin, TemplateView):...
266
267     class PasswordChangeView(PasswordContextMixin, FormView):...
290
291     class PasswordChangeDoneView(PasswordContextMixin, TemplateView):...
```

Abbildung 5.4.3: Klassen für die Aufgaben von EPIC 2

Dateipfad: apps/profiles/signals.py

```

69     publish_data = {
70         "username": str(user_data),
71         "logged_status": "True"
72     }
73     p = RabbitMq()
74     RabbitMq.publish(p, 'user_signed', publish_data)

```

Abbildung 5.4.4: Veröffentlichung des Anmeldungsstatus nach einer erfolgreichen Anmeldung im User-Service

Abbildung 5.4.5 zeigt die Daten, die veröffentlicht werden, wenn sich ein Benutzer abmeldet, was im Grunde innerhalb der Methode `BlacklistTokenUpdateView` passiert. Dies ist der Zeitpunkt, an dem das Token während des Abmeldevorgangs an die Blacklist gesendet wird.

Dateipfad: user_management/apps/manage_user/controller/user_login.py

```

        user_data.is_signed=False
    publish_data = {
        "username": str(user_data),
        "logged_status": str(user_data.is_signed)
    }
    p = RabbitMq()
    RabbitMq.publish(p, 'user_signed', publish_data)
    print("!!!!!! ", "shared logout message")
    user_data.save()
    return Response(status=status.HTTP_205_RESET_CONTENT)
except Exception as e:

```

Abbildung 5.4.5: Veröffentlichung des Anmeldungsstatus nach einer erfolgreichen Abmeldung im User-Service

EPIC 3: Benutzerkategorien und Suche

Wie in Abbildung 5.4.6 zu sehen ist, wurden die Storys aus EPIC 3 so implementiert, dass die Benutzer bzw. Tennisspieler nach einigen Eigenschaften gefiltert und nach bestimmten Kategorien abgefragt werden können.

Dateipfad: user_management/apps/manage_user/controller/user_user_search_and_category.py

```
25     # API to List Available Users
26     class FilterAvailableUsersAPIView(ListAPIView):...
27
28
29
30
31
32
33
34
35
36     # API to List users by strength
37     class FilterUsersStrengthAPIView(ListAPIView):...
38
39
40
41
42
43
44
45
46
47     # API to List user by ID
48     class FilterUsersByIDAPIView(ListAPIView):...
49
50
51
52
53
54
55
56     # API to List recommended players
57     class RecommendedPlayersAPIView(ListAPIView):...
58
59
60
61
62
63     # API to List latest players
64     class LatestPlayersAPIView(ListAPIView):...
65
66
67
68
69
70
71
72     # API to List popular players
73     class PopularUsersAPIView(ListAPIView):...
```

Abbildung 5.4.6: Klassen zu den Aufgaben von EPIC 3 in Initiative 1

EPIC 4: Profilverwaltung

Innerhalb der Profilverwaltung von EPIC 4 wird der Review-Service über eine Änderung informiert, wenn der `DestroyUserAPIView` in Zeile 167 oder der `DeleteAccount` in Zeile 194 aufgerufen wird. Dadurch wird eine Nachricht mit dem Property-Name `user_delete` gesendet, in der der Review-Service den Bewerter (als Rater bezeichnet) löscht, der diesem bestimmten gemeinsamen Benutzernamen entspricht.

Dateipfad: `user_management/apps/manage_user/controller/user_profile_management.py`

```
57     class UserAPI(generics.RetrieveAPIView):...
63
64
65     # API to fetch all registered users
66     class AllUsersAPIView(ListAPIView):...
67
68
69     # API to fetch User details by ID
70     class UserDetailView(APIView):...
71
72
73     # API to Post and Read Favorite Players
74     class WishListView(viewsets.ModelViewSet):...
75
76
77     # API to Update Profile
78     class UpdateProfileView(generics.UpdateAPIView):...
79
80
81     # API to Update a User's availability by User-ID
82     class UpdateAvailability(generics.UpdateAPIView):...
83
84
85     # API to Delete a User. Only admin is authorized.
86     class DestroyUserAPIView(DestroyAPIView):...
87
88
89     # API to Delete My own Profile.
90     class DeleteAccount(APIView):...
```

Abbildung 5.4.7: Klassen zu den Aufgaben von EPIC 4 in Initiative 1

Initiative 2: Review-Management

Im Rahmen des Review-Managements wird im folgenden Abschnitt nur eine Methode erörtert, die erklärt, wie das Problem der Benutzerauthentifizierung und der Feststellung, welche Berechtigung ein Benutzer hat, gelöst worden ist.

EPIC 1: CRUD von Bewertungen

In Abbildung 5.4.8 werden die Aufgaben der verschiedenen Klassen durch Kommentare verdeutlicht. Am interessantesten ist jedoch die Methode `create_opponent_review`, bei der das Problem der Authentifikation und Zugriffsrechte mit if-else-Anweisungen gelöst wird.

Dateipfad: review_service/apps/reviews/views.py

```
24     # API to Create a review
25     @api_view(["POST"])
26     @permission_classes([permissions.AllowAny])
27     def create_opponent_review(request, rater_username):...
90
91     # API to List all the reviews for a user by username
92     class UserRatingsView(APIView):...
119
120     # API to Read a review detail on <id>
121     class RatingDetailsView(APIView):...
137
138     # API to Delete a review on <id>
139     class DestroyReviewAPIView(DestroyAPIView):...
202
203     # API for updating review
204     class UpdateReviewAPIView(APIView):...
245
246     # API to List my own reviews
247     class MyReviewsAPIView(APIView):...
277
278     # API to Post (Reply) on a review ID of a specifc username
279     class ReplyToReviewAPIView(GenericAPIView):...
```

Abbildung 5.4.8: Klassen und Methoden zu den Aufgaben von EPIC 1 in Initiative 2

Dateipfad: review_service/apps/reviews/views.py

```
27 def create_opponent_review(request, rater_username): # profile_id is the url_param (the person
28     data = request.data
29     rater_user = get_object_or_404(Rater, username=rater_username) # is_signed=True,
30     rated_user = get_object_or_404(Rater, username=data['rated_user']) # id=rater_id
31
32     if rater_user.is_signed:
```

The screenshot shows a code editor with a dark theme. A yellow arrow points to the first line of the code. The code itself is a Python function named 'create_opponent_review'. It takes two parameters: 'request' and 'rater_username'. Inside the function, it retrieves 'rater_user' from the database using its username and sets the 'is_signed' attribute to True. It also retrieves 'rated_user' from the database using the 'rated_user' key from the 'data' dictionary and sets its 'id' attribute to 'rater_id'. Finally, it checks if 'rater_user' is signed in.

Abbildung 5.4.9: Überprüfung des Anmeldungsstatus eines Benutzers aus dem User-Service im Review-Service

Wie in den ersten vier Zeilen der Methode in Abbildung 5.4.9 zu sehen ist, wird der Benutzername des Bewerters aus dem URL-Parameter und der Benutzername des zu bewertenden Benutzers aus dem Body des zu bewertenden Benutzers bezogen. Die Informationen über den bewerteten Benutzernamen im Body, die an das Backend gesendet werden, entstammen der URL der zu bewertenden Benutzerseite und werden mit der Anfrage an den Review-Service-Backend gesendet.

Nachdem herausgefunden wurde, um welchen Bewerter es sich handelt, wird mit der Datenbank des Review-Service überprüft, ob dieser Benutzer angemeldet ist. Der Grund, warum der Review-Service Kenntnis darüber hat, dass dieser bestimmte Benutzer angemeldet ist, liegt in den Daten, die zuvor vom User-Service veröffentlicht wurden, um den Review-Service darüber zu informieren, dass der Benutzer sich gerade angemeldet hat; und dann wird das Attribut `is_signed` zu „True“. Sobald sich der Benutzer abmeldet, werden erneut Informationen über RabbitMQ ausgetauscht und das Attribut `is_signed` wird wieder auf „False“ gesetzt, so dass nur angemeldete Benutzer eine Bewertung abgeben können.

Die gleiche Logik wird auf andere Methoden innerhalb dieses Dienstes angewandt, um die Authentifizierung und Autorisierung bei Bedarf zu überprüfen.

5.5 View-Layer

Nach der Implementierung der View-Layer sieht das entwickelte Frontend im Frontend-Service fast identisch aus wie in den in Abschnitt 4.4 vorgestellten Abbildungen. Aus diesem Grund werden hier keine weiteren Abbildungen hinzugefügt. Es sollen nur einige zentrale Funktionen verdeutlicht werden.

In Abbildung 5.5.1 wird gezeigt, wie der Authorization-Header hinzugefügt wird. Zunächst wird versucht, das `access_token` hinzuzufügen, wobei davon ausgegangen wird, dass es sich nicht um ein JSON-Web-Token handelt, und wenn dies der Fall ist, wird ein JWT-Präfix vorangestellt. Das `access_token` wird mit `localStorage.setItem('access_token')` aus dem localStorage bezogen, wo es bereits während des Login-Prozesses gespeichert wurde. Diese `axiosInstance`-Methode wird jedes Mal verwendet, wenn ein autorisierter Zugriff vom Backend benötigt wird. Der Rest der Datei behandelt Fehler und setzt das `refresh_token` nach Ablauf des `access_token`.

Dateipfad: frontend_service/src/components/axios.js

```
5  const axiosInstance = axios.create({
6    baseURL: baseURL,
7    timeout: 5000,
8    headers: {
9      Authorization: localStorage.getItem('access_token')
10     ? 'JWT ' + localStorage.getItem('access_token')
11     : null,
12      'Content-Type': 'application/json',
13      accept: 'application/json',
14    },
15  });
16
```

Abbildung 5.5.1: Die Erstellung eines Authorization-Header mit einem JWT Präfix im Frontend-Service

Die Abbildungen 5.5.2 und 5.5.3 zeigen zwei Methoden, mit denen Daten aus dem Backend bezogen werden können. In der ersten Abbildung ist die Methode für den Abruf von Ressourcen zuständig, die keine Autorisierung benötigen, und es wird die aus der axios-Bibliothek importierte axios-Methode verwendet. In der zweiten Abbildung wird die `axiosInstance`-Methode verwendet, die in Abbildung 5.5.1 dargestellt wurde, um den Authorization-Header hinzuzufügen, wenn ein Zugriff auf autorisierte Ressourcen erforderlich ist. Zum Beispiel wird diese Methode verwendet, um den empfohlenen Benutzer abzurufen, wofür ein eingeloggter Benutzer erforderlich ist, der abgerufen werden muss.

Dateipfad: frontend_service/src/utils/AxiosURLGetter.js

```
6 | const tennisCompanionGetter = async (endpoint,
7 |                               getUsersState,
8 |                               setUsersState,
9 |                               getErrorMessage,
10 |                              setErrorMessage) => {
```

Abbildung 5.5.2: Eine Methode, um nicht autorisierte Ressourcen aus dem Backend zu erhalten

Dateipfad: frontend_service/src/utils/AxiosAuthURLGetter.js

```
7 | const authTennisCompanionGetter = async (endpoint,
8 |                               getUsersState,
9 |                               setUsersState,
10 |                              getErrorMessage,
11 |                             setErrorMessage) => {
```

Abbildung 5.5.3: Eine Methode, um autorisierte Ressourcen aus dem Backend zu erhalten.

In der Funktion `fetchUser` werden die mit den anderen oben genannten Funktionen abgerufenen Daten mittels der Funktion `map` iteriert und entsprechend auf der Seite für den Endbenutzer dargestellt.

Dateipfad: frontend_service/src/utils/UserFetcher.js

```
5 | export function fetchUsers (variableUserState, baseUrl) {
6 |
7 |   return variableUserState.users?.map((user, index) => {
8 |     return (
```

Abbildung 5.5.4: Eine Methode zur Iteration durch abgefragte Benutzer

Andernfalls werden der Login-Status und der Admin-Status auch mit dem Frontend geteilt und im localStorage gespeichert. Dieser wird dann in einem Hook oder einer Variable gespeichert, um die Daten nur für den richtigen Benutzer anzuzeigen, wie in den Abbildungen 5.5.5 und 5.5.6 gezeigt.

Dateipfad: frontend_service/src/user_management/user_searc_and_category/RecommendedPlayers.js

```
27 |   |   <div className="container mb-4">
28 |   |     | {userLoginStatus == 'true' &&
29 |   |     |   <>
```

Abbildung 5.5.5: Überprüfungs des Authentifizierungsstatus im Frontend-Service, um berechtigte Daten entsprechend anzuzeigen

Dateipfad: frontend_service/src/user_management/profile_management/admin/AdminDashboard.js

```
13 |   |   return userAdminStatus?
14 |   |     <>
```

Abbildung 5.5.6: Überprüfung des Autorisierungsstaus, um berechtige Daten entsprechend anzuzeigen

Kapitel 6

Testing

Die meisten Funktionen werden getestet, wobei Unit-Tests, Integrationstests und das Deployment von Tests mit Travis CI im Vordergrund stehen werden. Darüber hinaus werden Coverage Reports über die Tests mit einem pytest-cov Plugin erstellt.

6.1 Unit-Tests

Ein Unit-Test ist ein Verfahren zum Testen einer Unit – des kleinsten Codeteils, der in einem System logisch isoliert werden kann. Aus diesem Grund bestehen die Unit-Tests in diesem Projekt hauptsächlich aus den Tests für die models.py-Dateien. Ein Beispiel ist in Abbildung 6.1.1 zu sehen, wo der Benutzername und andere Attribute gemockt werden.

Dateipfad: user_management/tests/factories.py

```
4 ► +def test_user_str(base_user):...
7
8 ► +def test_user_short_name(base_user):...
12
13 ► +def test_user_full_name(base_user):...
17
18 ► +def test_base_user_email_is_normalized(base_user):...
22
23 ► +def test_superuser_email_is_normalized(superuser):...
27
28 ► +def test_superuser_is_not_staff(user_factory):...
34
35 ► +def test_superuser_is_not_superuser(user_factory):...
41
42 ► +def test_create_user_with_no_email(user_factory):...
```

Abbildung 6.1.1: Manche Unit-Tests im User-Service

6.2 Integration-Tests

Integrationstests kombinieren verschiedene Teile des Codes und der Funktionalität, um das Benutzerverhalten zu simulieren. Während mit einem Unit-Test sichergestellt werden kann, dass die Homepage einen HTTP-Statuscode von 200 erzeugt, kann mit einem Integrationstest der gesamte Registrierungsprozess eines Benutzers simuliert werden. Beim Integrationstest wird untersucht, wie die vielen Komponenten des Systems zusammenwirken, z. B. URL-Routing, Logik in views.py, Protokollierung und Abfrage von Modellen. (Humphrey 2021)

In einem typischen Integrationstest wird eine große Anzahl von Methoden getestet. Die Verwendung des ‚Django test client‘ ist ein eindeutiges Indiz für einen Integrationstest in Django. Zum Beispiel:

Dateipfad: user_management/tests/test_views.py

```
204 ► +    def test_logout(self):
205         c = Client()
206         c.force_login(self.user1)
207         url = reverse('apps.manage_user:blacklist')
```

Abbildung 6.2.1: Ein Beispiel für einen Integrationstest

6.3 Test-Coverage

Das Plugin Pytest-cov wird verwendet. Dieses Plugin erstellt ausführliche Coverage-Berichte über die Abdeckung aller Tests, die für die Anwendung durchgeführt wurden bzw. die noch übrig sind. In den beiden folgenden Abbildungen sind die Coverage-Berichte für den User- und den Review-Service dargestellt.

Module	statements	missing	excluded ↓	coverage
apps/common/admin.py	1	0	0	100%
apps/common/models.py	9	0	0	100%
apps/email_requests/admin.py	5	0	0	100%
apps/email_requests/models.py	13	1	0	92%
apps/manage_user/admin.py	17	0	0	100%
apps/manage_user/controller/profile_management.py	249	72	0	71%
apps/manage_user/controller/serializers/utils/decorators.py	9	0	0	100%
apps/manage_user/controller/user_login.py	62	37	0	40%
apps/manage_user/controller/user_register.py	16	0	0	100%
apps/manage_user/controller/user_search_and_categories.py	141	22	0	84%
apps/manage_user/managers.py	44	0	0	100%
apps/manage_user/models.py	36	0	0	100%
apps/manage_user/views.py	83	41	0	51%
apps/profiles/admin.py	7	0	0	100%
apps/profiles/exceptions.py	7	0	0	100%
apps/profiles/models.py	47	3	0	94%
apps/profiles/producer.py	31	11	0	65%
apps/profiles/renderers.py	9	4	0	56%
apps/profiles/signals.py	20	0	0	100%
user_management/api.py	0	0	0	100%
user_management/models.py	0	0	0	100%
Total	806	191	0	76%

Abbildung 6.3.1: Abdeckungsbericht über die abgeschlossenen Tests im User-Service

Dateipfad:

Module	statements	missing	excluded ↓	coverage
apps/reviews/admin.py	11	0	0	100%
apps/reviews/models.py	42	0	0	100%
apps/reviews/serializers.py	14	0	0	100%
apps/reviews/views.py	225	82	0	64%
Total	292	82	0	72%

Abbildung 6.3.2: Abdeckungsbericht über die abgeschlossenen Tests im Review-Service

6.4 Continuous-Integration

Travis CI

Travis CI führt jedes Mal, wenn Commits gepusht werden, Tests für ein GitHub-Repository durch. Travis CI ist ebenfalls ein gehosteter kontinuierlicher Integrationsservice, der zum Erstellen und Testen von Softwareprojekten auf GitHub und Bitbucket verwendet wird.

Die folgenden beiden anklickbaren Links führen zur Travis-CI Seite für den User-Service, sowie den Review-Service:

1. Travis-CI: User-Management-Service
2. Travis-CI: Review-Management-Service

Kapitel 7

Code-Dokumentationen

Neben den beschreibenden Funktionsnamen innerhalb des Projekts und der detaillierten Erläuterung einiger ihrer Funktionalitäten innerhalb dieses Projekts ist auch die Swagger-UI für eine bessere Ansicht der verfügbaren Endpunkte konfiguriert. Die Swagger-Seiten können über die folgenden beiden Pfade aufgerufen werden:

`http://localhost:8080/users/api/swagger/`

`http://localhost:8080/reviews/api/swagger/`

„Swagger UI ermöglicht es jedem - sei es Ihrem Entwicklungsteam oder Ihren Endverbrauchern - die API-Ressourcen zu visualisieren und mit ihnen zu interagieren, ohne dass die Implementierungslogik vorhanden ist. Sie wird automatisch aus Ihrer OpenAPI-Spezifikation (früher bekannt als Swagger) generiert, wobei die visuelle Dokumentation die Backend-Implementierung und die Nutzung auf der Client-Seite erleichtert.“(SwaggerDocu n.d.)

Darüber hinaus sind weitere Informationen zum Ausführen der Anwendung und andere notwendige Informationen in den README.md-Dateien der Repositories des Projekts hinterlegt, die unter den folgenden anklickbaren Links zu finden sind:

1. Repository: User-Management-Service
2. Repository: Review-Service
3. Repository: Frontend-Service

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Das Ziel dieses Projekts war es, das Tennis-Companion-Portal unter Verwendung der Microservices-Architektur zu erstellen. Zudem wurde angestrebt, eine Weiterentwicklung des Portals mit jeder anderen Programmiersprache zu ermöglichen. Um dies zu gewährleisten, wurden verschiedene Komponenten und Technologien benötigt. Einige dieser Technologien sind RabbitMQ, NGINX, Django, ReactJS, Docker, Docker Compose, Networks und Travis CI. Da die Vielzahl der Komponenten und die Kenntnisse, die für jede einzelne Komponente erworben werden müssen, zeitaufwendig sind, war es erforderlich, Prioritäten für einige Funktionen zu setzen, um am Ende dieser Arbeit ein funktionierendes Projekt zu erhalten. Aus diesem Grund war es unerlässlich, die MoSCoW-Methode anzuwenden, um die Aufgaben von den wichtigsten zu den unwichtigsten aufzulisten. Des Weiteren wurden User-Storys gesammelt und daraus wurde ein Use-Case-Diagramm abgeleitet, um eine klarere Vorstellung von den Aufgaben und den Anwendungsfällen zu bekommen, die für die Entwicklung des Portal gewünscht werden. Daraus wurden Initiativen und Epics dokumentiert, um die Aufgaben Schritt für Schritt vom Systementwurf und Konzeption bis hin zur Implementierung des Projekts zu durchlaufen.

Im Systementwurf wurden drei Schichten deklariert, die Modell-, die Controller- und die View-Layers, in denen die einzelnen Initiativen und Epics jeweils behandelt wurden. Wesentliche Diagramme des Systems als Ganzes wurden erstellt und einige erforderliche Attribute wurden in der Model-Layer dokumentiert. In der Controller-Layer wurden alle API-Endpunkte in verschiedenen Tabellen aufgelistet, die beschreiben, was jeder einzelne bewirkt. In der View-Layer wurden mit Adobe-XD Mock-ups erstellt, um einen Eindruck davon zu bekommen, wie das Portal nach der Entwicklung aussehen könnte.

Anschließend wurde in der Implementierungsphase die gleiche Logik angewandt, um mit der Umsetzung gemäß dem in der Design-Systementwurf-Phase erstellten Plan zu beginnen. Außerdem wurde die Konfiguration der im Projekt verwendeten Technologien anhand einiger Abbildungen demonstriert und relevante Informationen wurden verdeutlicht. In der Model-Layer wurde ein Entity-Relationship-Diagramm erstellt, und in der Controller-Layer wurden einige der zentralen Funktionen des Controllers diskutiert; dabei wurde auch erläutert, wie diese in Verbindung mit

RabbitMQ funktionieren. Die View-Layer hingegen wurde wie die geplanten Mockups implementiert.

Zudem wurden Tests erstellt, um zu sehen, ob die in den Projekten erstellten Views und Models erfolgreich und ohne Probleme getestet werden können, und es wurde ein Bericht über diese Tests erstellt, um einen Überblick über die Abdeckung zu erhalten.

Schließlich wurde das Projekt auf Github veröffentlicht, wo in der Datei ReadME.md Informationen darüber zu finden sind, wie das Projekt ausgeführt wird. Außerdem wurden alle relevanten Endpunktaufrufe mit Swagger dokumentiert.

Einige der geplanten Initiativen, wie die ‚Verwaltung von E-Mail-Request- und Timeslot-Management‘, konnten aus Zeitgründen nicht umgesetzt werden, aber die Umsetzung dieses Projekts ermöglicht es anderen, sich an der Entwicklung in der gewünschten Programmiersprache zu beteiligen. Sie müssen nur ihr Projekt mit RabbitMQ verbinden, einige notwendige Daten, wie den Benutzernamen der Benutzer, konsumieren und können anschließend mit der Programmierung beginnen.

8.2 Ausblick

Es gibt nicht viele Django-Projekte, die unter Verwendung der Microservices-Architektur entwickelt wurden, weshalb es eine Herausforderung war, dieses Projekt zu erstellen und ist damit eines der wenigen Django-Projekte, die mit Microservices entwickelt wurden. Die Schlussfolgerung ist jedoch, dass die Erstellung solcher Architekturen viel Zeit erfordert und die Komplexität während der Konfigurationsphase stark ansteigen kann. Sobald die Architektur jedoch konfiguriert ist, sinkt die Komplexität und die Entwicklung neuer Funktionen wird einfacher. Aus diesem Grund ist die Microservice-Architektur für kleine Projekte nicht optimal, da die Komplexität kurzfristig zunimmt. Ist das Projekt jedoch groß und muss mit der Zeit skaliert werden, ist die Komplexität am Anfang hoch und am Ende sehr gering. Aus diesem Grund kann es eine gute Lösung für die langfristige Perspektive sein und ein weiterer Vorteil ist die Unabhängigkeit zwischen den Services, d. h. wenn ein Service ausfällt, funktioniert der andere trotzdem. Dennoch hat das Projekt wegen dieser hohen Unabhängigkeit zwischen den Services noch so viel Potenzial, dass es weiter entwickelt werden kann. Zum Beispiel könnte Celery in Kombination mit RabbitMQ verwendet werden, um die Geschwindigkeit des Nachrichtenaustauschs zu erhöhen. Celery ist für asynchrone Operationen ausgelegt, d.h. Operationen werden in einem nicht-blockierenden Modus ausgeführt, so dass die Hauptoperation weiterlaufen kann. Es könnte auch ein temporärer Speicher eingerichtet werden, um alle gemeinsam genutzten Daten zu speichern, falls der RabbitMQ-Server zu irgendeinem Zeitpunkt ausfällt und andere Dienste während der Ausfallzeit wichtige Nachrichten verpassen könnten.

Ansonsten lassen sich darauf bereits neue Features mit neuen Datenbanken, neuen unabhängigen Attributen und Funktionalitäten aufbauen und alles was man braucht ist eine beliebige Programmiersprache und eine Verbindung zum RabbitMQ Server.

Eigenständigkeitserklärung

Ich erkläre hiermit, dass

- Ich die vorliegende wissenschaftliche Arbeit selbstständig und ohne unerlaubte Hilfe angefertigt habe,
- ich andere als die angegebenen Quellen und Hilfsmittel nicht benutzt habe,
- ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe,
- die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfbehörde vorgelegen hat.

Ort, Datum

Unterschrift

Literatur

- B, Akshay Kamath und Chaitra B H (18. Mai 2020). „A Comparison between RabbitMQ and REST ful API for Communication between Micro-Services Web Application“. In: *International Research Journal of Engineering and Technology (IRJET)* 7. ISSN: 2395-0056. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Credentials> (besucht am 1. März 2022).
- Bartlett, Jake (2016). *User Story vs Requirement – What’s The Difference?* URL: <https://blog.testlodge.com/user-story-vs-requirements/>.
- Buelta, Jaime (2019). *Hands-On Docker for Microservices with Python*. Birmingham: Packt Publishing.
- Clany, Molly (2021). *What Are Containers*. URL: <https://www.backblaze.com/blog/what-are-containers> (besucht am 6. Januar 2022).
- Djangodocu (2018). *Design philosophies*. URL: <https://docs.djangoproject.com/en/4.0/misc/design-philosophies/>.
- (2021). *Django Applications — Django Best Practices*. URL: <https://django-best-practices.readthedocs.io/en/latest/applications.html#loose-coupling>.
 - (n.d.). *Models*. URL: <https://docs.djangoproject.com/en/4.0/topics/db/models/>.
- Docker (2022). *Use containers to Build, Share and Run your applications*. URL: <https://www.docker.com/resources/what-container/> (besucht am 9. Januar 2022).
- Docker(FAQ) (2019). *Docker frequently asked questions (FAQ)*. URL: <https://docs.docker.com/engine/faq/#what-does-docker-technology-add-to-just-plain-lxc> (besucht am 2. März 2019).
- DockerDocu (2022a). *Developers Love Docker. Businesses Trust It*. URL: <https://www.docker.com/> (besucht am 9. Januar 2022).
- (2022b). *Dockerfile reference*. URL: <https://docs.docker.com/engine/reference/builder/>. (besucht am 10. Januar 2022).
- Farcic, Viktor (2014). *Continuous Deployment: Strategies*. URL: <https://www.javacodegeeks.com/2014/12/continuous-deployment-strategies.html>.
- Hochrein, Akos (2019). *Desgining Microservices with Django*. Berlin, Germany: Apress.
- Humphrey (2021). *Integration tests in Django*. URL: <https://www.codeunderscored.com/integration-tests-in-django/>.
- javatpoint (2022). *Django MVT*. URL: <https://www.javatpoint.com/django-mvt>.
- Jayanandana, Nilesh (2018). *Build a Docker Image just like how you would configure a VM*. URL: https://miro.medium.com/max/1400/1*p8k1b2DZTQEw_yfOhYniXw.png.

- Jr., Cloves Carneiro und Tim Schmelmer (2016). *Microservices from Day One: Build robust and scalable software from the start*. 1. Aufl. Apress Publishing.
- KnowHow (2021). *Dockerfile*. URL: <https://www.ionos.de/digitalguide/server/knowhow/dockerfile/> (besucht am 11. Januar 2022).
- Maureen, O'Gara (26. Juli 2013). „Ben Golub, Who Sold Gluster to Red Hat, Now Running dotCloud“. In: *DevOpsJournal*. URL: <https://web.archive.org/web/20190913100835/http://maureenogara.sys-con.com/node/2747331> (besucht am 9. Januar 2022).
- MDN (22. Februar 2022a). „Access-Control-Allow-Credentials“. In: *MDN web docs*. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Credentials> (besucht am 1. März 2022).
- (18. Februar 2022b). „Access-Control-Allow-Credentials“. In: *MDN web docs*. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Access-Control-Allow-Headers> (besucht am 1. März 2022).
- Noyes, Katherina (1. August 2013). „Docker: A ‘Shipping Container’ for Linux Code“. In: *Linux.com*. URL: <https://www.linux.com/news/docker-shipping-container-linux-code/> (besucht am 10. Januar 2022).
- Overflow, Stack (2020). *2020 Developer Survey*. URL: <https://insights.stackoverflow.com/survey/2020#overview> (besucht am 6. Januar 2022).
- Pautasso, Cesare u. a. (2017). „Microservices in Practice, Part 1: Reality Check and Service Design“. In: *IEEE Software* 34.1, S. 91–98. DOI: 10.1109/MS.2017.24.
- RabbitMQDocu (n.d.[a]). *RabbitMQ is the most widely deployed open source message broker*. URL: <https://www.rabbitmq.com/>.
- (n.d.[b]). *Which protocols does RabbitMQ support?* URL: <https://www.rabbitmq.com/protocols.html>.
- Ram, Shifani (17. Mai 2021). „A beginner’s guide: Building Front-End Applications with React“. In: *Frontend Weekly*. URL: <https://medium.com/front-end-weekly/a-beginners-guide-building-front-end-applications-with-react-1f0d3e75c0a7> (besucht am 9. Januar 2022).
- Research, Verified Market. (Februar 2020). „Cloud Microservices Market 2020 Trends, Market Share, Industry Size, Opportunities, Analysis and Forecast by 2026 – Instant Tech Market News“. In.
- Rupp, Chris und die SOPHISTen (2013). *Systemanalyse kompakt*. Nürnberg: Springer Vieweg.
- SwaggerDocu (n.d.). *Swagger UI*. URL: <https://swagger.io/tools/swagger-ui/>.
- W3C, Arbeitsvorlage des (2022). *Fetch*. URL: <https://fetch.spec.whatwg.org/> (besucht am 19. Februar 2022).
- Wirdemann, Ralf und Johannes Mainusch (2017). *Scrum mit User Stories*. 3. Aufl. HANSER.
- Wolfram, Hempel (2019). *Service Mesh VS API Gateway VS Message Queue - when to use what?* URL: <https://arcentry.com/blog/api-gateway-vs-service-mesh-vs-message-queue/>.
- XPhilosoph 212.144.156.154, Pm-sw u. a. (2020). *Anwendungsfall*. [Online; accessed 17-January-2021]. URL: <https://de.wikipedia.org/wiki/Anwendungsfall>.

Ziadé, Tarek (2017). *Python Microservices Development: Build, test, deploy, and scale microservices in Python*. 1. Aufl. Packt Publishing.