



Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Evaluierung von Lösungsansätzen zur Realisierung von Microservices mittels Python

Bachelorarbeit

Name des Studiengangs

Angewandte Informatik

Fachbereich 4

vorgelegt von

Maxim Irmscher

Datum:

Berlin, 06.02.2024

Erstgutachter: Herr Prof. Dr. Stephan Salinger

Zweitgutachter: Herr M.Sc. Tobias Dumke

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	1
1.3 Abgrenzung	2
1.4 Aufbau der Arbeit	2
2 Grundlagen	3
2.1 Monolith	3
2.2 Microservices	4
3 Methodik.....	5
3.1 Auswahl der Webframeworks	5
3.1.1 Vorauswahl	5
3.1.2 Auswahl	6
3.1.3 Django	7
3.1.4 Flask	7
3.1.5 FastAPI	7
3.1.6 Vergleichsobjekte	8
3.2 Vergleichskriterien.....	8
3.2.1 Skalierbarkeit.....	8
3.2.2 Wartbarkeit	9
3.2.3 Testunterstützung	9
3.2.4 Community Support.....	10
3.2.5 Dokumentation	10
3.3 Vergleich Django und FastAPI	11
3.3.1 Skalierbarkeit.....	11
3.3.2 Wartbarkeit	13
3.3.3 Testunterstützung	14
3.3.4 Community Support.....	15
3.3.5 Dokumentation	16

3.4 Fazit des Vergleichs.....	18
4 Anforderungsanalyse Beispielanwendung	19
4.1 Zielstellung.....	19
4.2 Anwendungsumgebung	19
4.3 Rahmenbedingungen / Technologien	20
4.4 Funktionale Anforderungen.....	21
4.5 Nicht-funktionale Anforderungen	22
4.6 Use Case Diagramme.....	23
5 Systementwurf.....	24
5.1 Entwurf Systemarchitektur	24
5.1.1 Umfrageverwaltung / Survey Service	25
5.1.2 Analyseservice / Analysis Service	25
5.1.3 API-Gateway.....	25
5.1.4 Nutzerverwaltung / User Service	26
5.2 Kommunikation	26
5.3 Data Model.....	27
5.3.1 User	27
5.3.2 Survey.....	28
5.3.3 Question	28
5.3.4 Response	28
6 Implementierung	29
6.1 Datenbankbindung.....	29
6.1.1 SQLAlchemy Model.....	29
6.1.2 Pydantic Schemas	30
6.2 HTTP APIs	32
6.2.1 Endpunktdefinition	32
6.2.2 Datenvalidierung.....	33
6.2.3 Exception Handling.....	33
6.2.4 API-Dokumentation.....	34

6.2.5 Dependency Injection	34
6.3 Nutzerverwaltung	35
6.3.1 Registrierung	35
6.3.2 Anmeldung	36
6.3.3 Autorisierung	37
6.4 Analyseservice	38
7 Tests	39
7.1 Unittest	39
7.2 Integrationstest	41
8 Demonstration und Evaluierung.....	42
8.1 Demonstration	42
8.2 Evaluierung	45
8.3 Herausforderungen	47
9 Ausblick und Zusammenfassung	48
9.1 Ausblick	48
9.2 Zusammenfassung	48
Abbildungsverzeichnis	A
Abkürzungsverzeichnis	C
Glossar	D
Quellenverzeichnis.....	E
Tabellenverzeichnis	H

1 Einleitung

1.1 Motivation

Aufgrund der fortschreitenden Digitalisierung werden Softwarelösungen zunehmend in unsere Alltagsbereiche integriert. Anwendungen sollen eine stetig wachsende Anzahl an Anforderungen erfüllen und werden infolgedessen kontinuierlich komplexer. Traditionell monolithische Architekturen stoßen hierbei häufig an ihre Grenzen, da sie unflexibel sowie schwierig zu skalieren und nur aufwendig zu warten sind.

Microservice-Architekturen – eine Vielzahl kleiner unabhängiger Dienste, die miteinander kommunizieren – bieten stattdessen einen vielversprechenden Lösungsansatz, um den Herausforderungen steigender Größe und Komplexität moderner Webanwendungen gerecht zu werden. Das Aufteilen der Gesamtfunktionalität in kleinere überschaubarere Einheiten verspricht mehr Flexibilität, Wartbarkeit und die Möglichkeit, Teile der Anwendung unabhängig voneinander zu skalieren.

Die Programmiersprache Python ist für ihre Vielseitigkeit und Nutzerfreundlichkeit bekannt, da sie eine Vielzahl an Bibliotheken zur Verfügung stellt und eine verständliche Syntax verwendet. In Kombination mit den Vorteilen einer Microservice-Architektur stellt Python eine gute Wahl dar, um den Herausforderungen moderner Softwareentwicklung gerecht zu werden.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, aktuell vielversprechende technische Lösungsansätze zur Umsetzung von Microservice-Architekturen, in Form von Python-Webframeworks, miteinander zu vergleichen. Der Vergleich soll anhand potentieller Kriterien wie u.a. Wartbarkeit und Skalierbarkeit mit dem Ziel durchgeführt werden, ein Webframework zu identifizieren, das die bestehenden Anforderungen an Microservices effizient, verständlich und technisch versiert zu erfüllen verspricht.

Im Anschluss an die Evaluation soll einer dieser Ansätze bei der Realisierung eines Softwareprojekts exemplarisch angewendet werden, um die zuvor herausgearbeiteten Vorteile praktisch zu erproben.

1.3 Abgrenzung

Der Fokus dieser Arbeit liegt auf der Evaluation Microservice-spezifischer Eigenschaften der untersuchten Python-Webframeworks. Auf die Bewertung von Gesamtfunktionalitäten wird bewusst verzichtet, um eine Verzerrung zugunsten von Fullstack-Lösungen gegenüber kleineren Ansätzen (einschließlich Microframeworks) zu vermeiden, insofern diese, ungeachtet ihrer Gesamtgröße und Funktionalität, vielversprechende Lösungsansätze bieten.

Der Anspruch der erstellten Beispielanwendung liegt nicht in der vollen Funktionsfähigkeit, sondern dient allein dem Zweck, die zuvor evaluierten Vorteile des ausgewählten Frameworks praxisnah zu testen. Die Implementierung verzichtet ferner auf datenschutzrelevante Maßnahmen und wird lokal bereitgestellt.

1.4 Aufbau der Arbeit

Einleitend werden die grundlegenden Konzepte von Monolithen und Microservice-Architekturen erläutert. Anschließend enthält das Kapitel Methodik eine Übersicht der gängigen Microservice-Frameworks unter Python, aus denen nach ausgewählten Kriterien jenes ermittelt wird, welches vielversprechende Lösungsansätze zur Implementierung einer auf Microservice-Architektur basierenden Webanwendung verspricht. Daraufgehend wird zunächst die Anforderungsanalyse für eine Beispielanwendung, durch die Umfragen generiert, beantwortet und ausgewertet werden können, durchgeführt und ein entsprechender Systementwurf erstellt. Im nächsten Schritt wird die Anwendung implementiert und dabei evaluiert, inwiefern das Arbeiten mit dem ausgewählten Framework den Erwartungen entspricht und welche Herausforderungen im Implementierungsprozess aufgetreten sind.

2 Grundlagen

In diesem Kapitel sind essenzielle Konzepte erläutert, die für das Verständnis der restlichen Arbeit von Bedeutung sind. Der Abschnitt „Monolith“ veranschaulicht das traditionelle Architekturmodell in sich geschlossener Anwendungen, die sämtliche Funktionalität der Geschäftslogik in einer einzigen Code-Basis in sich vereinen.

Im Anschluss ist das Konzept von Microservice-Architekturen dargestellt, bei dem große Anwendungen in viele kleine unabhängige Services aufgeteilt werden, die voneinander weitestgehend unabhängig agieren.

2.1 Monolith

In der Softwareentwicklung wird eine in sich geschlossene Anwendung, die dementsprechend unabhängig agiert, als „Monolith“ bezeichnet. Es handelt sich hierbei um ein eher traditionelles Architekturmodell, welches nach Harris [1] mit den Attributen „groß“, „schwerfällig“ und „gletscherartig“ in Verbindung gebracht wird und den Code sämtlicher Geschäftsprozesse, in einer einzigen Code-Basis, enthält. Der monolithische Architekturansatz bedingt nach Harris [1] folgende Vor- und Nachteile:

Das Entwickeln, Deployment, Testen und die Fehlerbehebung kleinerer Anwendungen (oder Projekten in ihrer Anfangsphase) gestaltet sich im Vergleich zu Microservice-Architekturen einfacher und schneller, da sich die gesamte Geschäftslogik innerhalb der gleichen Code-Basis befindet und nicht durch übergeordnete Infrastrukturen beeinflusst wird. Dieser Vorteil schwindet, sobald Anwendungen eine bestimmte Größe überschreiten und infolgedessen einen hohen Grad an Komplexität erreichen. Der Entwicklungsprozess wird verlangsamt, da die Geschäftslogik für den einzelnen Entwickler immer schwerer zu durchschauen ist. Selbst geringfügige Änderungen im Code erfordern stets das Kompilieren, Testen und Deployment der gesamten Code-Basis und können sich aufgrund fehlender Kapselung auf die restliche Anwendung unvorhergesehen auswirken. Monolithen sind oft durch die initiale Wahl der Technologie eingeschränkt und unflexibel, da sich nachträgliche Änderungen nur unter hohem Kosten- und Zeitaufwand realisieren lassen.

Zusammenfassend zeigt sich, dass der monolithische Architekturansatz mit Vorteilen bei der Entwicklung kleinerer Anwendungen einhergeht, jedoch mit zunehmender Größe und Komplexität erhebliche Herausforderungen hervorruft.

2.2 Microservices

Anwendungen, die auf einer Microservice-Architektur basieren, bestehen, im Gegensatz zu Monolithen, aus vielen kleinen voneinander entkoppelten Services mit eigener Code-Basis. Die Aufspaltung in kleinere Dienste verfolgt das Ziel, entstehende Nachteile großer monolithischer Anwendungen (siehe 2.1) auszugleichen und Gesamtkomplexität zu verringern. Die Geschäftslogik wird durch das Aufteilen überschaubarer und demzufolge leichter handhabbar. [1]

Nach Harris [1] liegt der Vorteil von Microservices darin, dass sowohl die Entwicklung als auch das Kompilieren, Testen, Deployment und Skalieren eines Microservices aufgrund seiner eigenen Code-Basis unabhängig von anderen Services erfolgen kann. Erreicht ein Dienst seine Lastkapazität, können problemlos neue Instanzen bereitgestellt werden. Bei der Nutzung von Clouddienstleistern bietet dies einen Vorteil, da viele Geschäftsmodelle oft nur die tatsächlich verbrauchte Rechenleistung in Rechnung stellen. Das Vorhalten und der damit verbundene Leerlauf von Servern für eventuelle Spitzenbelastung ist somit nicht notwendig.

Des Weiteren wird das Arbeiten in kleinen Teams gefördert und damit agile Arbeitsmethoden ermöglicht. Mitarbeiter können autonom experimentieren, da es jederzeit und ohne die Arbeit anderer zu beeinträchtigen, möglich ist, ein Rollback durchzuführen. Die Unabhängigkeit der Services ermöglicht darüber hinaus eine flexible Wahl und nachträgliche Anpassung der genutzten Technologien und vermindert die Gefahr, dass ein auftretender Fehler innerhalb eines Microservices den Absturz der gesamten Anwendung verursacht. Die Fehlervermeidung, -suche und -behebung ist in kleineren weniger komplexen Teilsystemen einfacher und macht die Anwendung insgesamt robuster. [1]

Die Verwendung von Microservice-Architekturen erfordert dafür einen hohen Grad an Organisation zwischen den Teams und kann beispielsweise bei unzureichender Koordination von Schnittstellen sowohl die Geschwindigkeit als auch Qualität der Entwicklung beeinträchtigen. Die Gesamtkosten einer Anwendung können sich zudem deutlich erhöhen, da pro Service oft eigene Test-Suiten, Hosting-Infrastrukturen und Überwachungswerkzeuge notwendig sind. Die korrekte Größe der Microservices und einen geeigneten Grad der Entkopplung zu ermitteln, erfordert einen hohen Initialaufwand und führt bei Fehlentscheidungen zu Problemen im späteren Entwicklungsprozess. [1]

Zusammenfassend bieten Microservices durch ihre Unabhängigkeit Vorteile in der Entwicklung und Skalierung. Für ihre effektive Nutzung erfordern sie jedoch eine sorgfältige Koordination zwischen den Teams sowie eine fundierte Planung, um potentielle Herausforderungen und erhöhte Kosten zu minimieren.

3 Methodik

Der Aufbau verteilter Anwendungen erfordert die Auswahl eines geeigneten Microservice-Frameworks. Im ersten Kapitel werden zunächst die Webframeworks ausgewählt, die die aktuell größte Unterstützung der Community bieten.

Anschließend sind entsprechende Vergleichskriterien festgelegt, die im Rahmen von Microservice-Architekturen eine bedeutende Rolle spielen, um daraufhin die Vergleichsobjekte nach diesen Maßstäben miteinander zu vergleichen. Ziel ist es, ein vielversprechendes Webframework für die Umsetzung einer Beispielanwendung auf Basis von Microservices zu identifizieren.

3.1 Auswahl der Webframeworks

Die Auswahl eines geeigneten Webframeworks zur Umsetzung einer beispielhaften Microservice-Architektur unter Python steht im Fokus dieses Kapitels. Es gilt zunächst eine allgemeine Liste relevanter Optionen aufzustellen, um aus dieser daraufhin die vielversprechendsten Kandidaten für einen detaillierten Vergleich zu filtern.

3.1.1 Vorauswahl

Die Aufnahme der folgenden Webframeworks in die Vorauswahl erfolgte aufgrund ihrer Präsenz in zahlreichen Onlineartikeln zum Thema Microservices mit Python. [2], [3], [4], [5]

- **Flask**
- **CherryPy**
- **Tornado**
- **Sanic**
- **Bottle**
- **Django**
- **FastAPI**
- **Pyramid**

3.1.2 Auswahl

Im professionellen Umfeld ist es wichtig, schnell fundierte Unterstützung bei auftretenden Problemen zu erhalten, um einen effizienten Entwicklungsprozess zu ermöglichen. Die meisten Herausforderungen treten erfahrungsgemäß jedoch nicht zum ersten Mal auf und wurden durch die Community bereits an anderer Stelle gelöst. Portale wie GitHub und Stack Overflow bieten Lösungsvorschläge, um Bugs zu identifizieren oder Hinweisen zur Codeoptimierung.

Im Folgenden wird anhand des GitHub Star Vergleichs auf star-history.com die Größe der Community und den damit erwartbaren Support genauer beleuchtet, um die Liste potentiell geeigneter Webframeworks entsprechend einzugrenzen.

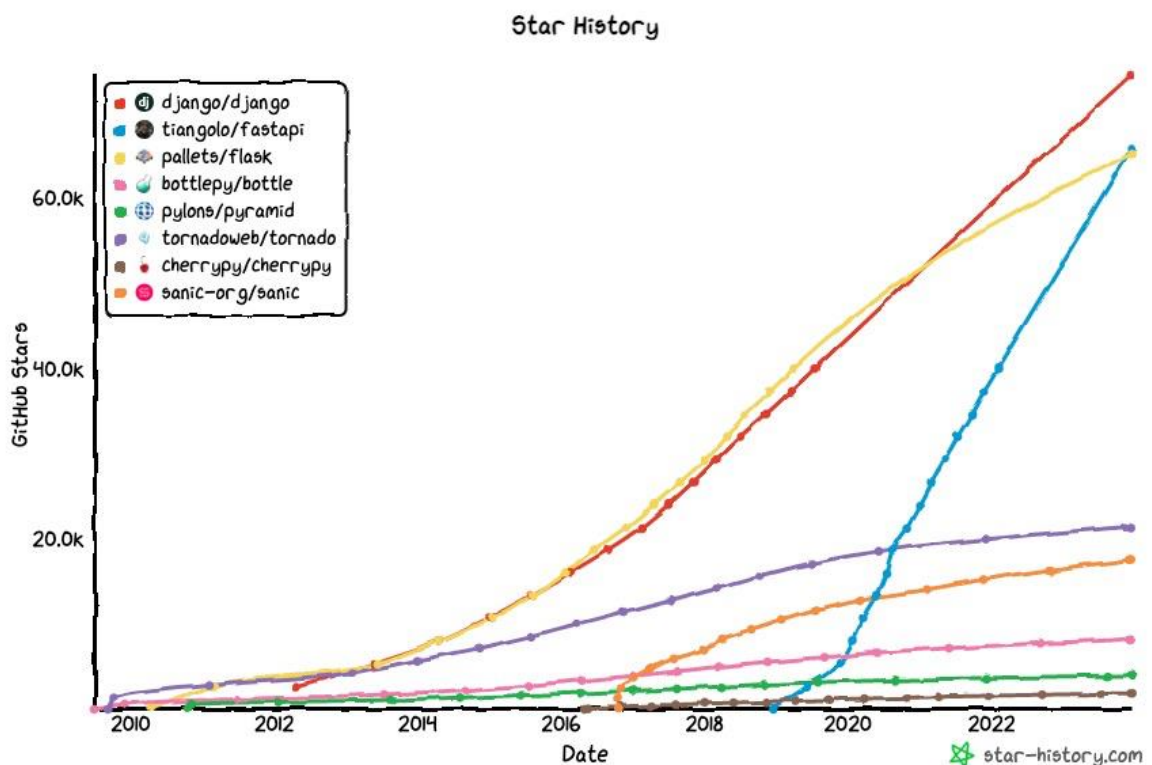


Abbildung 1: Vergleich GitHub Stars [6]

Aus diesem Vergleich geht hervor, dass sich zum Stand 14.12.2023 die Webframeworks **Django** (74.500), **FastAPI** (65.800) und **Flask** (65.200) auf GitHub sehr großer Beliebtheit in der Community erfreuen. Tornado (21.400), Sanic (17.500), Bottle (8.200), Pyramid (3.900) und CherryPy (1.700) werden aufgrund Ihrer signifikant niedrigeren Popularität aus dem weiteren Vergleich ausgeschlossen.

3.1.3 Django

Django ist ein kostenfreies Fullstack Open-Source-Framework, das von Adrian Holovaty und Simon Willison entwickelt wurde, um den Entwicklungsprozess von Webanwendungen sowohl zu vereinfachen als auch zu beschleunigen. Beide arbeiteten im Jahr 2003 als Webentwickler bei der Zeitung „*Lawrence Journal-World*“ und standen, aufgrund im Journalismus üblicher kurzer Veröffentlichungsfristen, unter großem Druck, schnell neue Features sowie vollständige Webanwendungen zu entwickeln. Diese Situation hat beide motiviert, Django zu entwickeln und im Jahre 2005 als Open-Source Software zu veröffentlichen. Seitdem wurde es von der Open-Source Community kontinuierlich weiterentwickelt und verbessert. Es bietet Lösungen für viele Probleme, vor denen die Entwickler von Django selbst einmal standen. Django hat sich seitdem zu einem beliebten Webframework entwickelt, und wird von Diensten wie Instagram, National Geographic und Pinterest verwendet. [7], [8]

3.1.4 Flask

Flask ist ein Mikro-Webframework, das im Jahre 2010 von Armin Ronacher veröffentlicht wurde. Flask verfolgt einen stark minimalistischen Ansatz und wird aufgrund seiner Schlantheit oft bei der Entwicklung kleinerer Projekte und Prototypen genutzt. Es bietet lediglich die Template-Engine Jinja (die ebenfalls in Django integriert ist) und das WSGI-Toolkit „Werkzeug“. Es verzichtet auf andere Funktionalitäten wie Authentifizierung, Datenbankbindung und Caching, da diese bereits durch andere Bibliotheken umgesetzt wurden, und sich einfach in Flask integrieren lassen. [9], [10], [11]

3.1.5 FastAPI

FastAPI, genutzt von Unternehmen wie Uber und Netflix, wurde von Sebastián Ramírez mit Hilfe von Pydantic – einer Python-Bibliothek zur Datenvalidierung – entwickelt und im Jahre 2018 als Open-Source Webframework veröffentlicht. Die Motivation dafür entsprang der Notwendigkeit, dass zum Entwickeln komplexer Programmierschnittstellen (APIs) oft eine Vielzahl verschiedener Werkzeuge und Plug-ins notwendig ist. Ziel war es, eine Lösung zu schaffen, die die besten Werkzeuge und Ideen bereits bestehender Lösungsansätze wie u.a. Django und Flask in sich kombiniert. Der Fokus von FastAPI liegt im Erstellen von API-Endpunkten, die von modernen Frontend Frameworks wie VueJS oder React genutzt werden. Zusätzlich bietet es das automatische Generieren einer API-Dokumentation. [12], [13], [14]

3.1.6 Vergleichsobjekte

Ziel ist es, Webframeworks in ihrem eigentlichen Umfang zu vergleichen und externe Bibliotheken, insofern diese nicht offensichtlich zum jeweiligen Ökosystem gehören, aus dem Vergleich auszuschließen. Der folgende Vergleich wird sich dementsprechend auf **Django** und **FastAPI** beschränken, da Flask ein sehr minimalistisches Framework ist und einen ähnlichen Funktionalitätsumfang nur durch das Einbinden externer Bibliotheken erreicht.

3.2 Vergleichskriterien

In diesem Kapitel sollen geeignete Vergleichskriterien festgelegt werden, um danach beide Frameworks nach diesen miteinander zu vergleichen. Der Vergleich orientiert sich dabei an Kriterien der Norm ISO/IEC 9126 (Qualitätskriterien für Software als Produkt) und legt besonderen Fokus auf Eigenschaften, die mit Microservice-Architekturen im Zusammenhang stehen.

3.2.1 Skalierbarkeit

Ein großer Vorteil von Microservices gegenüber monolithischen Anwendungen besteht darin, dass sich einzelne Services bei steigenden Nutzerzahlen und der damit einhergehenden zunehmenden Last unabhängig voneinander skalieren lassen – das Hinzufügen und Entfernen von Ressourcen ist je nach aktuellem Bedarf individuell möglich.

Einerseits kann dies, unabhängig vom gewählten Webframework, auf Architekturebene geschehen, bei dem externe Autoscaler die aktuelle Last ermitteln und im Zusammenspiel mit Containersystemen wie Kubernetes neue Instanzen benötigter Dienste hochfahren. [15]

Andererseits ist eine Verbesserung der Skalierbarkeit auch auf Anwendungsebene mit Hilfe verschiedener, Framework-spezifischer Mechanismen möglich. So stellt eine effiziente Datenvalidierung die Bearbeitung ausschließlich korrekter Anfragen sicher, um im Ergebnis die Gefahr von Bottlenecks zu verringern. Eine asynchrone API ermöglicht das gleichzeitige Verarbeiten von Anfragen, ohne die Anwendung bis zum Erhalt einer Antwort zu blockieren, und trägt damit ebenfalls zu einer besseren Skalierbarkeit bei. [16]

Dieser Vergleich konzentriert sich auf Funktionalitäten, die das Framework auf Anwendungsebene zur Verfügung stellt.

3.2.2 Wartbarkeit

Steigende Nutzerzahlen korrelieren, neben der Notwendigkeit horizontaler Skalierung, oft auch mit vertikalem Wachstum durch neue oder sich ändernde Anforderungen.

Eine wartbare Anwendung ist in der Lage, flexibel auf sich ändernde Umstände zu reagieren. Sie ermöglicht es, neue Services nahtlos zu integrieren und vorhandene Dienste entsprechend der sich ändernden Anforderungen – wie das Integrieren neuer Features oder das Schließen von Sicherheitslücken – anzupassen, ohne dass dies erheblichen Aufwand oder Unterbrechungen verursacht.

Ein Microservice-Framework kann dazu beitragen, eine Webanwendung wartbar und damit zukunftssicher zu gestalten, indem es beispielsweise eine schnelle Entwicklung oder die einfache Integration unterschiedlicher Technologien ermöglicht.

3.2.3 Testunterstützung

Die Qualitätssicherung durch Tests ist unerlässlich, um Fehler in einer Anwendung präventiv zu verhindern und deren zuverlässige Funktionsweise unter verschiedenen Bedingungen zu gewährleisten.

Durch eine umfassende Testabdeckung wird sichergestellt, dass die Anwendungslogik in verschiedenen Szenarien zuverlässig und wie erwartet funktioniert. Ein Webframework sollte dementsprechend eine breite Palette an verschiedenen Testarten wie u.a. Unit- und Integrationstests unterstützen.

Des Weiteren sind geeignete Mocking-Werkzeuge hilfreich, um das Verhalten externer Dienste oder anderer Komponenten zu simulieren und somit die Korrektheit der Logik einzelner Bausteine isoliert zu testen.

Ein Framework, das diese Aspekte gut unterstützt, erleichtert das Entwickeln zuverlässiger und qualitativ hochwertiger Microservices.

3.2.4 Community Support

Die Größe und Aktivität der jeweiligen Entwicklergemeinschaft wird hier genauer analysiert, um im Ergebnis das Webframework zu ermitteln, bei dem ein vielversprechender Support und damit einhergehend effizienter Entwicklungsprozess zu erwarten ist. Eine engagierte Community bietet eine Fülle an Wissen zu bereits bewährten Lösungen sowie den Zugang zu weitreichenden Erfahrungsschätzen und Best Practices und deutet auf stetige Weiterentwicklung und Fehlerbehebung hin.

Um sicherzustellen, dass das Framework nicht nur die aktuellen Bedürfnisse unterstützt, sondern auch zum langfristigen Erfolg eines Projektes beiträgt, ist die Analyse des Community Supports von entscheidender Bedeutung.

3.2.5 Dokumentation

Die offizielle Dokumentation ist für Entwickler beim Erlernen eines neuen Frameworks oder Auftreten von Herausforderungen im Entwicklungsprozess oft der erste Anlaufpunkt. Eine qualitativ hochwertige Dokumentation ist daher entscheidend, um eine bestmögliche Unterstützung und Orientierung zu bieten.

Sie sollte die derzeitigen Funktionalitäten umfassend abdecken und die kontinuierlichen Veränderungen stets aktuell abbilden. Das Verwenden verständlicher Sprache, Aufteilen in überschaubare Abschnitte und Erläutern von Fachbegriffen hilft den Entwicklern, die entsprechenden Informationen leicht zu erfassen und ist somit für einen reibungslosen Entwicklungsprozess unverzichtbar. Beispiele und Tutorials sind hilfreich, da diese die Funktionsweise eines Frameworks praktisch veranschaulichen und den Lernprozess zur effektiven Nutzung unterstützen.

Eine gut strukturierte und aktuelle Dokumentation hat einen dementsprechend großen Einfluss auf die reibungslose Verwendung und ist in der Entwicklung von Webanwendungen ein wertvolles Werkzeug.

3.3 Vergleich Django und FastAPI

In diesem Kapitel wird mithilfe der zuvor genannten Kriterien ein detaillierter Vergleich zwischen Django und FastAPI vorgenommen, um im Ergebnis ein Webframework auszuwählen, das sich gut zur Umsetzung einer Microserviceanwendung eignet.

3.3.1 Skalierbarkeit

Python-Anwendungen werden auf einer Vielzahl verschiedener Webserver bereitgestellt, die sich beispielsweise durch die Wahl des darunterliegenden Betriebssystems in ihrer Konfiguration voneinander unterscheiden. Die individuelle Anpassung an den jeweiligen Webserver ist dabei oft mit hohem Aufwand verbunden.

Zur Vereinfachung dieses Prozesses wurde im Jahre 2003 das Python Web Server Gateway Interface (WSGI) definiert. Es handelt sich dabei um ein Protokoll, das auf einem Applikationsserver implementiert und neben dem eigentlichen Webserver auf dem gleichen physischen oder virtuellen Server mit installiert wird. Das WSGI fungiert als Vermittler zwischen Anwendung und Webserver und trägt die Verantwortung, HTTP-Anfragen sowie deren entsprechende Antworten weiterzuleiten. Die Anwendung kommuniziert nun ausschließlich mit dem WSGI-Server, ohne dass eine individuelle Anpassung an jeden Webserver notwendig ist. [17], [18]

Die Kommunikation zwischen Applikations- und Webserver funktioniert im Falle von WSGI synchron: Jede eintreffende Anfrage wird sequentiell abgearbeitet. Die Bearbeitung weiterer Anfragen ist bis zur entsprechenden Antwort oder einem eventuellen Time-Out blockiert. Bei großen Anwendungen mit vielen gleichzeitigen Anfragen kann dies zu Leistungsengpässen führen.

Im Jahr 2015 startete daraufhin die Entwicklung des Asynchronus Server Gateway Interface (ASGI) Protokolls. ASGI ermöglicht sowohl synchrone als auch asynchrone Beantwortung von Anfragen und ist rückwärtskompatibel zu WSGI. Es wurde im Jahre 2019 veröffentlicht. [19]

Django ist ein Webframework, das mithilfe eines WSGI-Servers bereitgestellt wird. Infolgedessen erfolgt die Abarbeitung von API Anfragen unabhängig von der Anwendungslogik ausschließlich sequentiell. Um dieser Einschränkung entgegenzuwirken, unterstützt Django seit der Einführung der Version 3.0 im Jahre 2019 nun auch ASGI und damit die asynchrone Bearbeitung von Anfragen.

Die Verwendung von asynchronem Code erweist sich dennoch als kompliziert und umständlich, da Django ursprünglich für den synchronen Gebrauch entworfen wurde. So gibt es keine standardmäßige Unterstützung für asynchrone Operationen im Django Rest Framework. Aufgaben wie Validierung, Bereinigung von Nutzereingaben sowie das Überprüfen von Berechtigungen sind manuell und ohne die Hilfe von Klassenoperationen zu implementieren. Darüber hinaus ist es erforderlich die Cross Site Request Forgery (CSRF) Middleware zu deaktivieren, um die asynchronen API Endpunkte benutzen zu können. Dies birgt ein zusätzliches Sicherheitsrisiko. [20], [21]

Django beinhaltet dennoch eingebaute Features wie ein eigenes Object-relational mapping (ORM), integrierte Caching Mechanismen und ein integriertes Admin Interface, die das Warten und Skalieren von Anwendungen unterstützen.

FastAPI hingegen basiert auf Starlette – einem ASGI-Framework – und ist dementsprechend auf die Verwendung von Asynchronität optimiert. Es hat sich im Kern auf das Erstellen von Microservices spezialisiert und bietet zahlreiche zusätzliche Funktionen, mit der Intention das Implementieren einer Microserviceanwendung zu vereinfachen und zu beschleunigen.

Die Integration externer ORM Bibliotheken wie SQLAlchemy und Tortoise-ORM ist nutzerfreundlich gestaltet und innerhalb der FastAPI-Dokumentation hinreichend erläutert. Im Gegensatz zu Django lässt FastAPI allerdings ein eigenes Caching-System vermissen. Das Einbinden externer Services wie Redis bietet sich dabei als gute Alternative an.

Funktionell steht FastAPI mit Unterstützung gut integrierter, dokumentierter und zum Ökosystem gehörender Bibliotheken mit der All-in-One Lösung von Django auf einer Stufe. Beide Frameworks eignen sich zum Implementieren einer skalierbaren Anwendung, wobei FastAPI durch seinen modernen, auf Asynchronität und Microservice spezialisierten Aufbau durch schnellere Entwicklungszeiten und nutzerfreundlichere Handhabung im Vorteil zu sein scheint.

3.3.2 Wartbarkeit

Django orientiert sich in seinem Aufbau am MVC-Pattern (Model-View-Controller), wobei Unterschiede hinsichtlich der Terminologie und der spezifischen Aufteilung von Verantwortlichkeiten bestehen. So wird Django oft als MVT-Framework (Model-View-Template) bezeichnet. Das Model und dessen Funktionalität bleibt hierbei unverändert. Die Template-Schicht übernimmt die eigentlichen Verantwortlichkeiten der View-Schicht des MVC-Patterns und ist als Präsentationsschicht dafür verantwortlich, was beim Nutzer angezeigt wird. Die View-Schicht übernimmt im MVT die Verantwortlichkeiten des Controllers. Die klare Trennung der Verantwortlichkeiten sorgt für eine Entkopplung und unterstützt die Wartbarkeit insofern, dass später einzelne Komponenten ausgetauscht, erweitert oder verändert werden können.

Das in Django integrierte ORM unterstützt die Austauschbarkeit der im Hintergrund laufenden Datenbank, ohne dass die Notwendigkeit besteht, manuell SQL zu schreiben und diesen Code bei eventuellem Austausch der Datenbank zu aktualisieren. Ein Nachteil besteht darin, dass es lediglich SQL-Datenbanken unterstützt. Die Wartbarkeit wird dadurch insofern eingeschränkt, dass zur Nutzung von NoSQL-Datenbanken statt des integrierten ORMs das Verwenden externer Bibliotheken erforderlich ist. [22], [23]

FastAPI erzwingt im Gegensatz zu Django keine strikte Trennung zwischen Model, View und Controller und setzt stattdessen auf eine flexible Strukturierung des Codes. Es bietet den Entwicklern die Möglichkeit, neben dem MVC-Pattern auch andere Architekturansätze zu verfolgen. Im Gegensatz zu Django enthält FastAPI kein eigenes ORM. In Zusammenarbeit mit Pydantic (Datenvalidierung durch Modelle) und SQLAlchemy (Datenbankrepräsentation) ist dieses dennoch unkompliziert integrierbar und bietet zusätzlich entsprechenden Support für NoSQL-Datenbanken.

Sowohl Django als auch FastAPI legen gleichermaßen großen Wert auf gute Codestrukturierung, wobei FastAPI deutlich leichter zu erlernen ist und für Anwendungen, die auf Microservice-Architekturen und das Erstellen von APIs zur Kommunikation zwischen diesen aufgebaut sind, maßgeschneidert wirkt. FastAPI ist moderner und erscheint in Bezug auf Architektur deutlich flexibler, da beispielsweise das MVC-Pattern angewendet werden kann, aber nicht wie bei Django implizit erforderlich ist. Beide Frameworks bieten eine unkomplizierte ORM Integration, ein deklaratives URL-Routing und im Falle von FastAPI Dependency Injection, welche zur Lesbarkeit und damit Wartbarkeit des Codes beitragen.

3.3.3 Testunterstützung

Django bietet ein integriertes Testframework, das auf dem Python-Standardmodul „unittest“ basiert und Entwicklern damit Zugriff auf alle Funktionalitäten dieser Bibliothek ermöglicht. Es beinhaltet eine umfassende Bandbreite an Funktionalitäten wie einen Test-Runner zur Ausführung der Tests, Assertions zum Abgleich erwarteter und tatsächlicher Ergebnisse, Mocking für die Simulation bestimmter Objekte und Funktionalitäten sowie Test-Discovery zur automatischen Identifikation der Testklassen, um die Organisation des Test-Codes zu erleichtern.

Ein Test-Client, der als Dummy Web Browser fungiert und das Testen von Views durch das Simulieren von HTTP-Requests unterstützt, die Klasse SimpleTestCase – eine Subklasse von unittest.TestCase – mit erweiterter Funktionalität (wie das Verifizieren der Gleichheit von URLs) und Fixtures mit der Aufgabe Datenbanken vor Testausführung mit Dummy Daten zu befüllen, sind ebenfalls Bestandteil des Django Testframeworks.

Zusammenfassend bietet Django mit seiner umfassenden Bandbreite an integrierten Testwerkzeugen eine geeignete Unterstützung bei der Entwicklung einer robusten, zuverlässigen und wartbaren Anwendung. [24], [25], [26]

FastAPI enthält kein eigenes Testframework. Da es allerdings auf Starlette basiert und Starlette wiederum die HTTPX-Bibliothek – eine HTTP-Client Bibliothek für Python – im eigenen Test-Client benutzt, um HTTP-Anfragen gegen die Anwendung zu testen, ist das Testen von APIs komfortabel gelöst. Bibliotheken wie „unittest“ und „pytest“ ermöglichen das Mocking, bieten Assertions und Fixtures. [27], [28]

Im Testen der Anwendung sind sowohl Django als auch FastAPI sehr gut aufgestellt und lassen keine Funktionalität vermissen. Der Hauptunterschied besteht darin, dass Django die gesamte Funktionalität bereits fest im eigenen Testframework integriert hat. FastAPI bedient sich hingegen der Funktionalität von Starlette, unittest und bei Bedarf externer Bibliotheken wie Pytest. Die Integration ist in der offiziellen Dokumentation beschrieben und stellt sich als unkompliziert heraus.

3.3.4 Community Support

GitHub Sterne

Die Anzahl der GitHub Sterne ist zwar kein direkter Indikator für die Qualität eines Produkts und spiegelt auch nicht die Anzahl der tatsächlichen Nutzer wider. Dennoch gibt uns diese Kennzahl eine grobe Auskunft über die Popularität und das allgemeine Interesse an einem Repository. So spricht eine hohe Anzahl an Sternen dafür, dass das Projekt von vielen Entwicklern genutzt wird und großes Vertrauen der Community genießt.

Wie bereits im Kapitel 3.1.2 festgestellt, liegt **Django** mit **~74.500** leicht vor **FastAPI** mit **~65.800**. Beide befinden sich auf einem ähnlichen Niveau wie bekannte Webframeworks anderer Sprachen (beispielsweise **Spring-Boot** mit **~70.900** GitHub Sternen [29]).

Package Downloads

Package Downloads sind, ähnlich der Anzahl an GitHub Sternen, eine geeignete Kennzahl, um Annahmen über die Größe der Community und Aktualität eines Frameworks zu treffen:

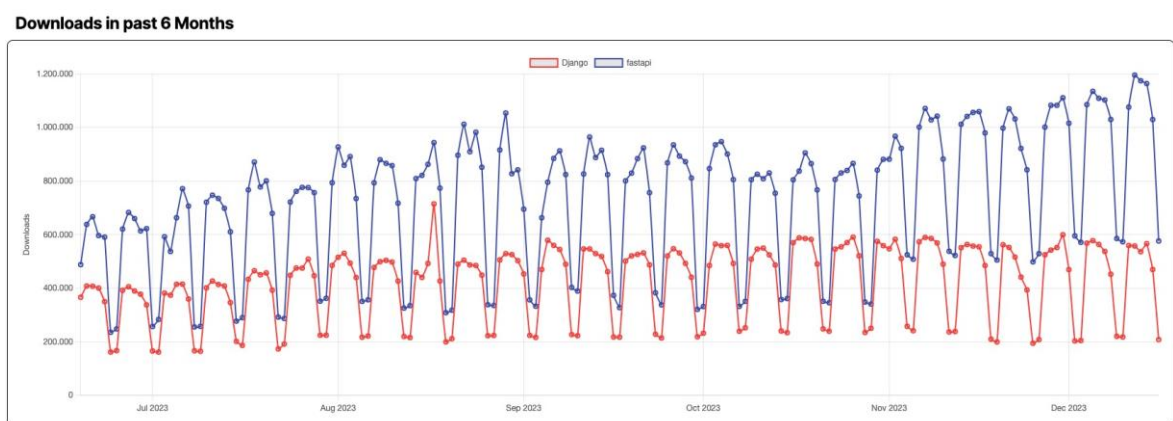


Abbildung 2: Vergleich Package Downloads Django und FastAPI [30]

In Abbildung 2 ist zu sehen, dass sich die Anzahl der Downloads von FastAPI im Zeitraum der letzten 6 Monate von ca. 600.000 auf mittlerweile 1.200.000 verdoppelte, während sich Downloads von Django von 400.000 auf nun 600.000 nur um ca. 50% erhöht haben. Hierbei ist zu beachten, dass Django bereits im Jahre 2005 veröffentlicht wurde und damit schon seit 17 Jahren verfügbar ist. FastAPI hingegen gibt es erst seit 5 Jahren und erfreut sich seitdem wachsender Beliebtheit.

Google Trends

Google Trends wird genutzt, um die Häufigkeit der Suchabfragen bestimmter Suchbegriffe zu analysieren und miteinander zu vergleichen. Dies bietet neben den bereits erwähnten Methoden Auskunft über die Größe der Community und gibt Hinweise auf deren weitere Entwicklung.

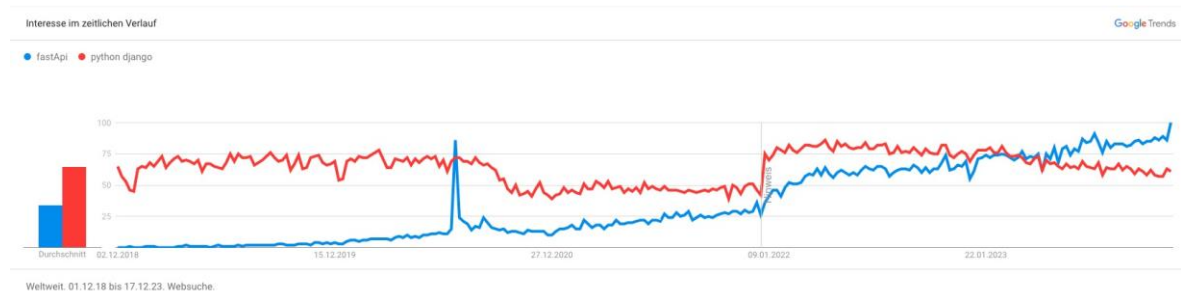


Abbildung 3: Vergleich Google Trends Django und FastAPI [31]

Die Häufigkeit der Suchanfragen zum Thema FastAPI nimmt, wie in Abbildung 3 zu sehen, kontinuierlich zu und hat seit März 2023 einen höheren Anteil als die Suchanfragen zum Thema Django. Ein steigendes Interesse der Community gegenüber dem Newcomer FastAPI im Vergleich zum alteingesessenen Django ist hier deutlich erkennbar.

Dennoch ist diese Statistik mit Bedacht zu betrachten, da der Suchbegriff „Django“ aufgrund des gleichnamigen Spielfilms „Django Unchained“ aus dem Jahre 2012, der die Suchergebnisse verfälscht, auf „python django“ eingegrenzt wurde.

Die Statistiken zeigen, dass sowohl bei Django als auch FastAPI ein ausgezeichneter Community Support zu erwarten ist. Während bei Django aufgrund seiner längeren Verfügbarkeit schon viele Herausforderungen bewältigt wurden, ist bei FastAPI ein außerordentlich rasant steigendes Interesse der Community zu beobachten.

3.3.5 Dokumentation

Django bietet eine klar strukturierte und umfassende Dokumentation mit einsteigerfreundlichem Tutorial, die Schritt für Schritt am praktischen Beispiel die wichtigsten Konzepte (u.a. Requests und Responses, Models, Views und Templates) erläutert.

Das Tutorial enthält die Topic Guides, die grundlegende Konzepte wie das Handling von HTTP Requests, Testing, Caching, Logging und viele weitere Themen besprechen, Reference Guides, die die technischen Prinzipien erläutern nach denen Django funktioniert (beispielsweise werden alle beim File Handling relevanten Klassen mit deren Funktionen oder der Aufbau von Exceptions detailliert beschrieben) und How-To Guides zu Themen wie Deployment oder dem Generieren von PDF Files.

Die Dokumentation enthält jeweils eigene Kategorien für die Verwendung von Modellen zur Datenbankintegration (ORM), Anfragenverarbeitung und dazu, wie Templates für die Präsentation zu verwenden sind. Sie spiegelt damit das Model View Template Prinzip von Django wider.

Neben einer Suchfunktion, Links zu Online Communities (wie dem offiziellen Forum oder einem Discord Server) und einem FAQ Bereich ist die Dokumentation in neun verschiedenen Sprachen verfügbar und wird regelmäßig aktualisiert, wobei ältere Versionen jederzeit zu erreichen sind. [32]

Django bietet sehr tiefgründige und umfangreiche Informationen, die Entwickler beim Verwenden dieses Frameworks unterstützt. Sie setzt dabei im Vergleich zu anderen Dokumentationen allerdings viel Vorwissen voraus.

Ähnlich wie bei Django enthält auch die Dokumentation von FastAPI einen umfangreichen Tutorial-Abschnitt, der dem Nutzer Schritt für Schritt und dabei sehr verständlich die Kernfunktionen (wie das Erstellen von APIs, die automatisch generierte Dokumentation der Endpunkte und Security-Themen wie OAuth2) an praktischen Beispielen näherbringt.

Die Dokumentation enthält darüber hinaus eigene Abschnitte zur automatischen Validierung von Anfragen sowie asynchroner Programmierung – beides Kernkonzepte von FastAPI. Der How-To Bereich erläutert die interne Funktionsweise. Eine eigene Deployment Sektion erklärt detailliert, was Deployment bedeutet und welche Strategien FastAPI dabei verfolgt. [33]

Beide Dokumentationen sind übersichtlich gegliedert und beschreiben detailliert und umfangreich sowohl die Kernfunktionen als auch weitere verwendete Konzepte der jeweiligen Frameworks.

FastAPI geht dabei ein wenig behutsamer mit dem Leser um, da es verständlichere Sprache verwendet, weniger Wissen voraussetzt und gerade für Neueinsteiger aufgrund der geringeren Größe weniger überwältigend erscheint (Django ist insgesamt deutlich umfangreicher, was sich entsprechend im Umfang der Dokumentation niederschlägt).

3.4 Fazit des Vergleichs

Die Gesamtbetrachtung zeigt, dass beide Frameworks sehr gut zur Implementierung einer Microserviceanwendung geeignet sind, wobei FastAPI im Vergleich zu Django einen moderneren und auf Microservices spezialisierteren Eindruck erweckt.

FastAPI nutzt das ASGI und wurde von Anfang an mit dem Konzept asynchroner Programmierung und der damit einhergehenden effizienteren Kommunikation zwischen Microservices entwickelt. Django hingegen unterstützt zwar inzwischen ASGI, legt allerdings im Entwicklungsprozess asynchroner Anwendungen den Entwicklern auf Grund seiner Wurzeln in WSGI große Steine in den Weg. [21]

Auch wenn beide Frameworks gleichermaßen Wert auf gute Codestruktur legen, so bietet FastAPI gegenüber Django die Vorteile, nicht strikt an eine MVC-Architektur gebunden zu sein, neben SQL auch NoSQL Datenbanken zu unterstützen und ist durch seine flachere Lernkurve, automatisch generierter API-Dokumentation, integrierter Dependency Injection, intuitiver Annotationen und leicht verständlicher Online-Dokumentation deutlich einfacher zu benutzen.

Einen großen Vorteil bietet Django aufgrund seiner integrierten Template Engine Jinja, die eine separate Frontendanwendung ersetzt und damit Komplexität verringert. Dieser Vergleich soll sich allerdings auf größere, im professionellen Rahmen erstellte Webanwendungen fokussieren, bei denen stattdessen oft eigene Frontends mithilfe von Frameworks wie Angular, React oder VueJS erstellt werden, um die einzelnen Verantwortlichkeiten im Sinne von Microservice-Architekturen weiter zu entkoppeln.

In Anbetracht der gewonnenen Erkenntnisse zeigt sich FastAPI im Rahmen dieser Arbeit als interessante Option für die geplante Anwendung, da es ähnlich gute Ergebnisse wie das altbewährte Fullstack-Framework Django verspricht, dies aber aufgrund seines auf Microservice spezialisierten Aufbaus deutlich effizienter, schneller, intuitiver und insgesamt nutzerfreundlicher zu erreichen verspricht. Insofern FastAPI diesen Erwartungen gerecht wird, läge der Erkenntnisgewinn darin, dass zum Implementieren einer modernen auf Microservice-Architektur basierenden Anwendung nicht zwingend ein großes vollumfängliches Webframework wie Django notwendig ist. [33]

4 Anforderungsanalyse Beispielanwendung

Das folgende Kapitel umfasst die Anforderungsanalyse der Beispielanwendung. Zunächst wird die Zielstellung definiert und anschließend die Umgebung festgelegt, unter der die Anwendung bereitgestellt wird. Anschließend gilt es, die Rahmenbedingungen und Technologien festzulegen, unter und mit Hilfe derer die Implementierung durchgeführt wird. Zuletzt folgt die Definition funktionaler und nicht-funktionaler Anforderungen.

4.1 Zielstellung

Das Ziel besteht darin, eine Beispielanwendung auf Grundlage einer Microservice-Architektur unter der Verwendung von FastAPI zu entwickeln, die es Anwendern ermöglicht, Umfragebögen zu erstellen und diese von anderen Nutzern beantworten zu lassen. Die Ergebnisse einer Umfrage sollen als Diagramme dargestellt und in üblichen Formaten (CSV, PDF) zur weiteren Auswertung bereitgestellt werden können. Die Anwendung soll drei verschiedene Nutzerrollen (Admin, Editor, Respondent) sowie folgende Hauptfunktionalitäten unterstützen:

- **Nutzerverwaltung:** Registrierung, Authentifizierung, Autorisierung
- **Umfrageverwaltung:** Erstellen, Beantworten und Löschen von Umfragen
- **Analyse:** Grafische Darstellung von Umfrageergebnissen
- **Datenexport:** Export der Umfrageergebnisse als PDF und CSV
- **GUI:** Grafische Benutzeroberfläche zur Interaktion mit der Anwendung

4.2 Anwendungsumgebung

Die zu entwickelnde Anwendung ist ein exemplarisches Projekt und dient ausschließlich dem Zweck der Überprüfung der zuvor herausgearbeiteten Vorteile des Webframework FastAPI im Kontext von Microservice-Architekturen. Um einen entsprechenden Testrahmen zu schaffen, wird sie lokal bereitgestellt und ist über einen Webbrowser zugänglich. Auf öffentliche Bereitstellung wird verzichtet.

4.3 Rahmenbedingungen / Technologien

Die Beispielanwendung soll im Ergebnis folgenden Rahmenbedingungen entsprechen und die jeweils festgelegten Technologien zur Umsetzung nutzen.

Architektur

Die Anwendung basiert auf einer Microservice-Architektur und besteht dementsprechend aus verschiedenen eigenständigen Diensten, die weitestgehend unabhängig bereitgestellt und skaliert werden können. Jeder Microservice hat einen spezifischen Verantwortungsbereich mit klar definierten Schnittstellen, über die mit anderen Diensten kommuniziert wird.

Backendtechnologie

Das Backend ist unter der Verwendung des Webframeworks FastAPI, inklusive der zum Ökosystem gehörenden Bibliotheken implementiert. Dazu zählen beispielsweise Pydantic zur Datenvalidierung und Serialisierung, SQLAlchemy für die Datenbankintegration oder unittest zur Implementierung von Tests.

Kommunikation

Die Kommunikation zwischen den Microservices erfolgt mithilfe von RESTful APIs, die den Austausch von Informationen zwischen Diensten in einem standardisierten JSON Format über HTTP ermöglichen.

Frontendtechnologie

Um die Anwendung nutzerfreundlich bedienen zu können, ist mit Hilfe des Frameworks VueJS beispielhaft ein Frontend implementiert.

Datenbankintegration

Die Umfragen mit den entsprechenden Umfrageergebnissen sowie Nutzerprofile werden in einer PostgreSQL Datenbank persistiert.

4.4 Funktionale Anforderungen

Die Anwendung soll folgende funktionale Anforderungen erfüllen:

Nutzerverwaltung

Die Nutzerverwaltung stellt eine Möglichkeit zur Registrierung von Anwendern bereit, deren Profile daraufhin in einer Datenbank persistiert werden. Nach erfolgreicher Registrierung wird eine Token-basierte Authentifizierung durchgeführt, um sicherzustellen, dass ein angemeldeter Nutzer entsprechend seiner zugewiesenen Rolle autorisiert ist, auf entsprechende Ressourcen zuzugreifen.

Berechtigungen

Das System unterscheidet zwischen drei verschiedenen Nutzerrollen, die jeweils spezifische Berechtigungen besitzen. Die Administrator-Rolle hat die Befugnis, Nutzerprofile zu verwalten. Dies beinhaltet das Löschen von Nutzerprofilen inklusive der dazugehörigen Umfragen sowie deren Ergebnisse. Die Editor-Rolle darf eigene Umfragen erstellen, löschen und sich die grafische Auswertung anzeigen lassen. Nutzer mit der Rolle „Respondent“ sowie anonyme Nutzer sind berechtigt, Umfragen zu beantworten.

Verwaltung der Umfragen

Die Umfrageverwaltung beinhaltet das Erstellen, Beantworten und Löschen von Umfragen mit unterschiedlichen Fragestellungen mithilfe von Freitextfeldern, Checkboxes und Dropdown-Menüs. Die Umfragen werden in einer separaten Datenbank persistiert. Umfrageergebnisse können als Torten- und Balkendiagramm dargestellt werden. Der Export von Umfrage- und Nutzerlisten ist im CSV und PDF Format möglich.

Nutzeroberfläche

Die Anwendung verfügt über eine einfache grafische Nutzeroberfläche, die es den Anwendern ermöglicht, mit ihr zu interagieren. Sie soll funktional und nutzerfreundlich gestaltet sein.

4.5 Nicht-funktionale Anforderungen

In der Beispielanwendung wird ein besonderer Fokus auf die folgenden nicht-funktionalen Qualitätsmerkmale gelegt, da diese den Schwerpunkt des zuvor durchgeführten Vergleichs bilden und sich speziell auf die Vorteile von Microservice-Architekturen beziehen. Im Anschluss an die Implementierung wird evaluiert, wie FastAPI speziell mit diesen Anforderungen umgeht.

Skalierbarkeit

Die Anwendung ist so implementiert, dass die Skalierung einzelner Microservices unabhängig voneinander möglich ist, um so den Herausforderungen erhöhter Last gerecht zu werden.

Wartbarkeit

Die Anwendung ist möglichst wartbar gestaltet. Der Code ist klar strukturiert und gut lesbar. Die einzelnen Services sind derart voneinander entkoppelt, dass Änderungen in einem Service die Funktionalität eines anderen entsprechend wenig beeinflussen.

Testabdeckung

Die einzelnen Microservices sind beispielhaft durch entsprechende Unit- und Integrationstests getestet, um die gewollte Funktionalität innerhalb eines Services zu garantieren. Eine vollständige Testabdeckung ist im Rahmen dieses Projekts nicht notwendig, da das Ziel lediglich darin besteht, beispielhaft zu prüfen, wie problemlos verschiedene Testarten und Testfälle mit FastAPI abgedeckt sind.

API-Dokumentation

Die Anwendung stellt eine API-Dokumentation bereit, um eine unkomplizierte Interaktion mit anderen Services (und im professionellen Rahmen mit anderen Teams) zu begünstigen.

4.6 Use Case Diagramme

In diesem Kapitel sind die wichtigsten Funktionalitäten der einzelnen Akteure durch Use-Case-Diagramme dargestellt.

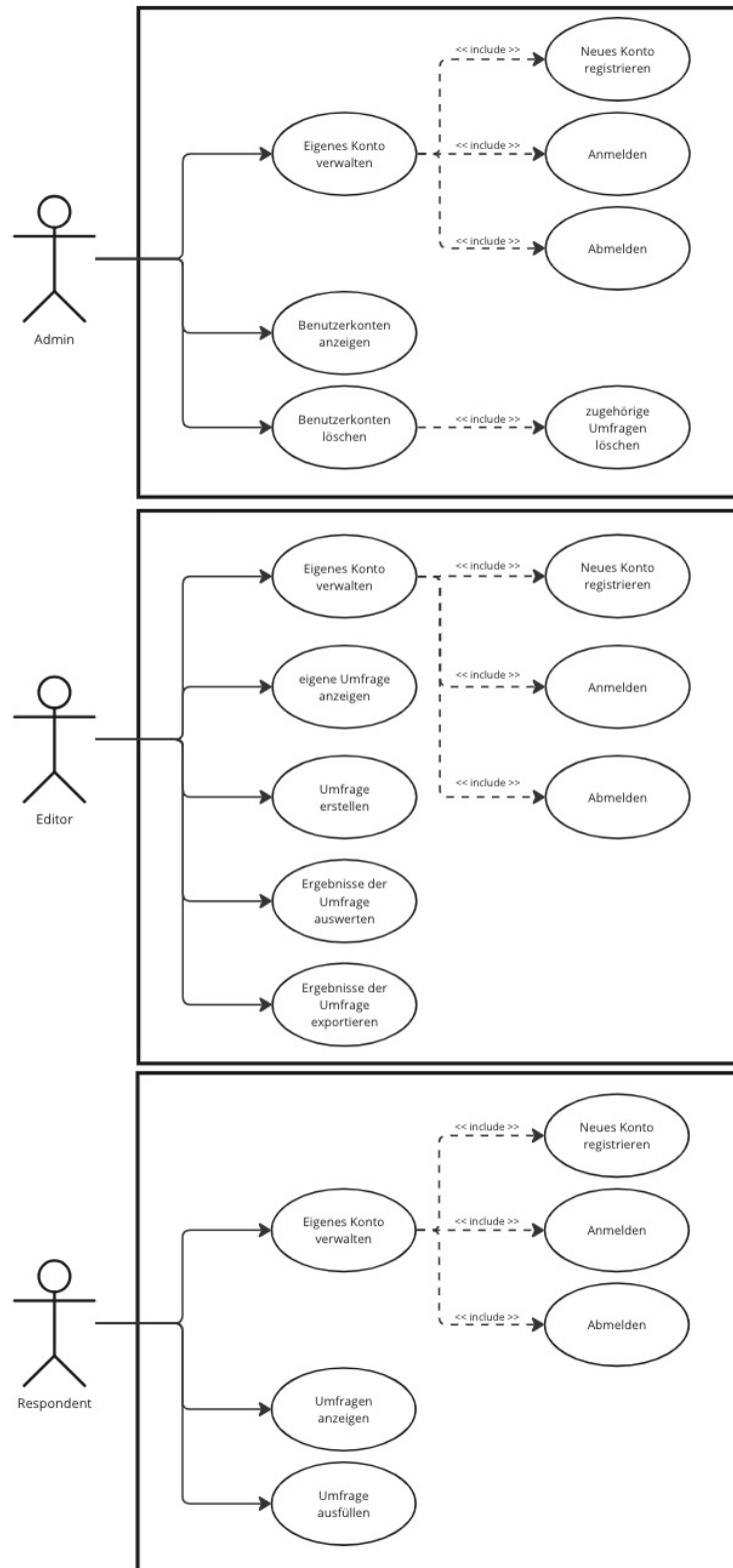


Abbildung 4: Use Case Diagramme

5 Systementwurf

Das folgende Kapitel dient dazu, die zuvor festgelegten Anforderungen an die Beispielanwendung in eine konkrete Systemarchitektur zu überführen. Die Verantwortlichkeiten werden dabei in verschiedene Microservices aufgeteilt und deren jeweiliger Verantwortungsbereich definiert. Anschließend wird festgelegt, wie die einzelnen Dienste miteinander kommunizieren, um eine reibungslose Zusammenarbeit zu gewährleisten.

Dieses Kapitel beschreibt darüber hinaus die notwendigen Datenmodelle und enthält schematische Darstellungen des geplanten Frontend-Designs.

5.1 Entwurf Systemarchitektur

Die folgende Abbildung visualisiert die geplante Systemarchitektur und dient als Leitfaden für die geplante Entwicklung der Beispielanwendung.

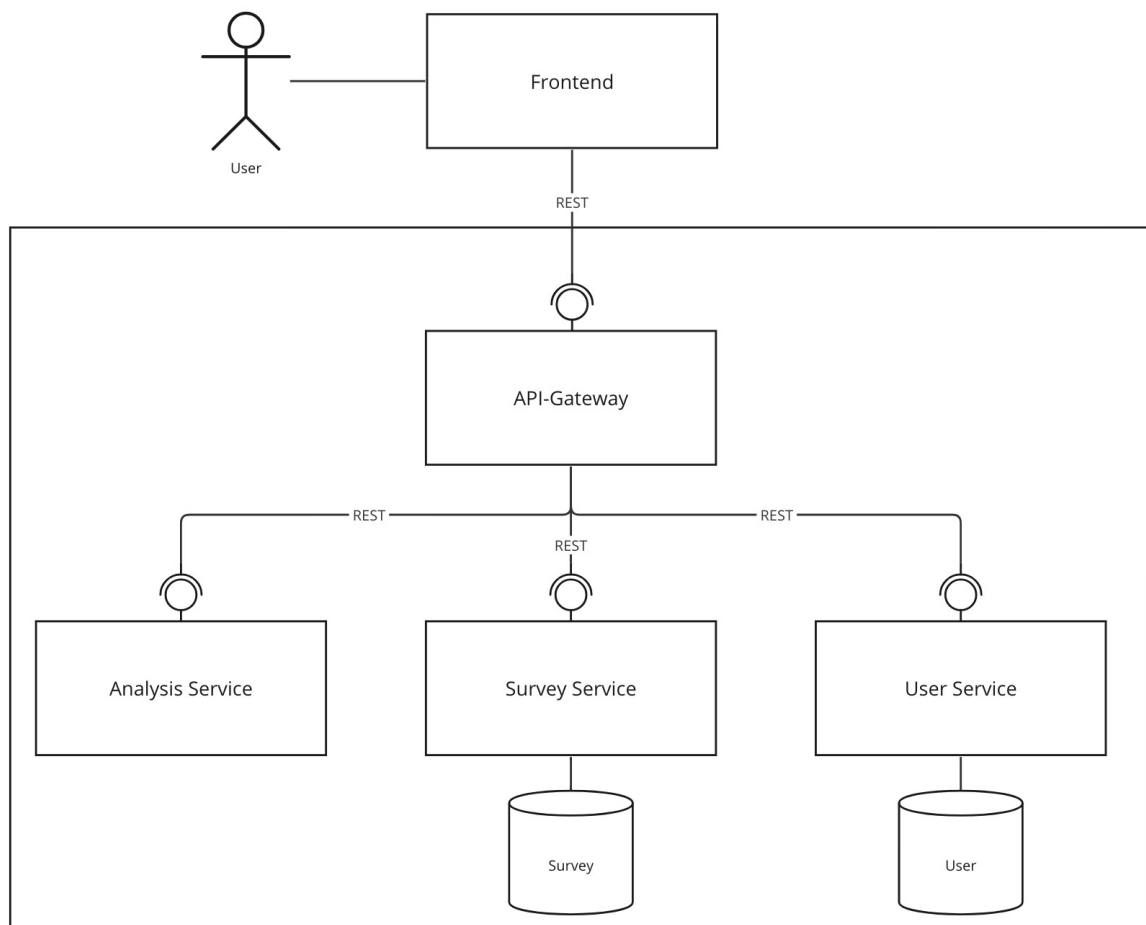


Abbildung 5: Entwurf Systemarchitektur

5.1.1 Umfrageverwaltung / Survey Service

Die Umfrageverwaltung übernimmt alle Aufgaben, die mit der Verwaltung von Umfragen in Verbindung stehen. Dazu gehört das Persistieren neu erstellter Umfragen inklusiver dazugehöriger Antworten und Herausgeben sowie Löschen gespeicherter Daten.

Sie beinhaltet hauptsächlich CRUD-Operationen, wird über einen REST-Controller angesprochen und benötigt eine Datenbankanbindung, wodurch die ORM Anbindung getestet wird.

5.1.2 Analyseservice / Analysis Service

Der Analyseservice dient als Beispiel eines Microservices, der losgelöst vom eigentlichen Kerngeschäft (Verwaltung von Umfragen) eine eigene Geschäftslogik enthält, keine Datenbank benötigt und im realen Umfeld gut erweiterbar oder austauschbar ist.

Die Aufgabe dieses Services besteht darin, bei Erhalt einer Anfrage eine Auswertung der vorhandenen Antworten vorzunehmen, und das ausgewertete Ergebnis zurückzusenden.

Die Ergebnisse sind so aufzubereiten, dass das Frontend diese in einem geeigneten Format erhält und selbst keine Berechnungen mehr durchführen muss.

In einem größeren Rahmen wäre an dieser zu überlegen, ob ein eigener Caching-Mechanismus oder eine entsprechende Datenbank integriert werden soll, um bei großer Last Antwortzeiten zu verringern.

5.1.3 API-Gateway

Das API-Gateway dient als Vermittlungsinstanz zwischen der Frontendanwendung und dem restlichen Microservice-System. Das Frontend hat für alle Anfragen das API-Gateway als einzigen Ansprechpartner, welches die Anfragen intern an die entsprechenden Services vermittelt und deren Zusammenarbeit orchestriert. Somit genügt dem Frontend das Wissen, wie es mit dem Gateway zu kommunizieren hat, ohne die genaue interne Struktur der Microservice-Architektur zu kennen. Dies sorgt für eine bessere Entkopplung und erhöht damit die Wartbarkeit der gesamten Anwendung.

Eine weitere Aufgabe des Gateways besteht darin, dass Verantwortungen wie Authentifizierung und Autorisierung mithilfe des User Services zentral an einer Stelle erfolgen und keine separate Prüfung in jedem Service erforderlich ist. Die einzelnen Dienste können sich so auf ihre Kernkompetenz konzentrieren und davon ausgehen, dass jede Anfrage vom Gateway bereits auf entsprechende Berechtigungen überprüft wurde.

5.1.4 Nutzerverwaltung / User Service

Die Nutzerverwaltung übernimmt die Verwaltung (Erstellen, Persistieren, Authentifizieren, Ändern und Löschen) von Nutzerkonten. Authentifizierung meint hier das Überprüfen der vom User eingegebenen Anmeldeinformationen, um sicherzustellen, dass diese mit den Daten des im System gespeicherten Nutzerkontos übereinstimmen. Bei erfolgreicher Anmeldung wird ein Token erzeugt, dieses zurückgegeben und im Konto des jeweiligen Nutzers persistiert. Erhält dieser Service eine Autorisierungsanfrage vom API-Gateway mit einem Token, ist zu prüfen, inwiefern dieses einem Nutzer zugeordnet ist. Im Erfolgsfall sind die entsprechenden Daten an das API-Gateway zu senden, damit dieses dazu in der Lage ist, die Autorisierung anhand der Rolle durchzuführen.

5.2 Kommunikation

Die Kommunikation der einzelnen Services untereinander erfolgt via REST. Dies bietet die Möglichkeit, FastAPI auf seine Fähigkeit zu testen, REST-Controller zu entwickeln und entsprechende API-Dokumentationen zu generieren.

5.3 Data Model

Das nachfolgend definierte Data Model (siehe Abbildung 6) modelliert das Erstellen von Umfragen, die aus verschiedenen Fragen mit unterschiedlichen Fragetypen (wie Freitext oder Multiple Choice) bestehen. Befragte sowie anonyme User können anschließend auf diese Fragen antworten. Die Auswertung und Präsentation einzelner Umfrageergebnisse wird ebenfalls durch dieses Datenmodell unterstützt.

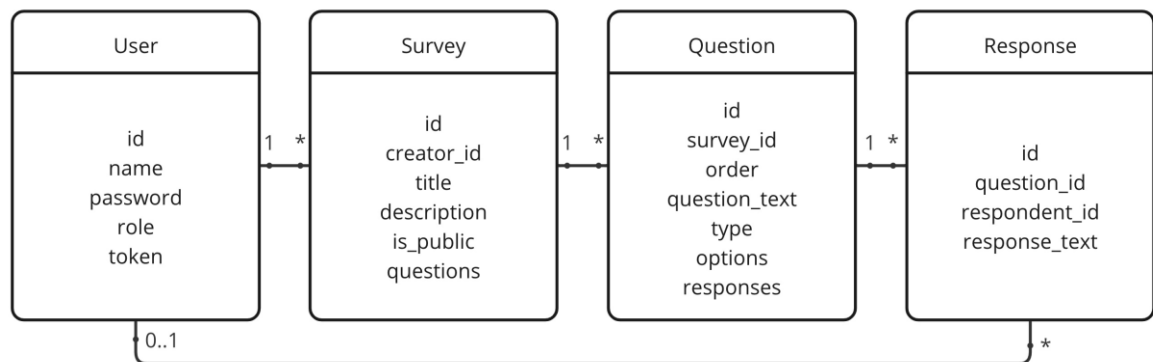


Abbildung 6: UML-Klassendiagramm Data Model

5.3.1 User

Die im Nutzer (User) gespeicherten Daten dienen der Identifikation und Autorisierung eines Anwenders. Er benötigt eine eindeutige ID, um die Umfragen später einem Nutzer zuordnen zu können sowie Name und Passwort zur Anmeldung in der Anwendung. Die Rolle und das Token dienen der Autorisierung bzw. Überprüfung, welche Berechtigungen der jeweilige Nutzer besitzt.

- id: einzigartiger Identifier zur Identifikation eines Nutzers
- name: Name des Nutzers
- password: Passwort des Nutzers
- role: Rolle des Nutzers (Admin, Editor, Respondent)
- token: einzigartiger Identifier zur Autorisierung eines Nutzers

5.3.2 Survey

Bei der Umfrage (Survey) handelt es sich um ein Gerüst, das die zu einer Umfrage gehörenden Einzelfragen bündelt.

- id: einzigartiger Identifier zur Identifikation der Umfrage
- creator_id: Fremdschlüssel zum Nutzer, der die Umfrage erstellt hat
- title: Name der Umfrage
- description: Beschreibung der Umfrage
- is_public: Sichtbarkeit der Umfrage

5.3.3 Question

Die Einzelfrage (Question) beinhaltet neben zur Identifikation notwendigen IDs den Typ der Frage, die Ordnungsnummer, die eigentliche Fragestellung und Antwortoptionen:

- id: einzigartiger Identifier zur Identifikation der Einzelfrage
- survey_id: Fremdschlüssel zur Umfrage
- order: Zuordnung der Reihenfolge innerhalb der Umfrage
- question_text: Fragestellung der Einzelfrage
- type: Art der Frage (Radio-Button, Check-Box, Freitext)
- options: (optionale) Antwortmöglichkeiten für Radio-Buttons, Check-Boxes

5.3.4 Response

Die Antwort (Response) enthält ein Textfeld, das die Antwort auf Freitextfragen enthält und eine Liste der ausgewählten Antwortmöglichkeiten bei Fragen mit Dropdown-Menü oder Checkboxes. Das Feld „respondent_id“ ist im Falle einer anonymen Beantwortung null.

- id: einzigartiger Identifier zur Identifikation der Antwort
- question_id: Fremdschlüssel zur Einzelfrage
- respondent_id: Fremdschlüssel zum Nutzer, der die Frage beantwortet hat.
- Response_text: Antworttext auf Freitextfragen / ausgewählte Antworten bei Dropdown-Menü und Check-Boxes

6 Implementierung

Die Implementierung der Anwendung erfolgt in einem iterativen Prozess, wobei jeder Service unabhängig voneinander entwickelt wird, um den Grad der Kopplung möglichst gering zu halten und damit die Wartbarkeit und Skalierbarkeit zu erhöhen. Der Startpunkt ist dabei das Anbinden der Datenbank an die Umfrageverwaltung. Hierbei werden zunächst die Datenmodelle übertragen, danach die erforderlichen CRUD Operationen erstellt und zuletzt die zugehörigen REST-Endpunkte definiert. Das Implementieren der Nutzerverwaltung erfolgt im Anschluss und folgt dem gleichen Schema der Umfrageverwaltung.

Anschließend sind Analyseservice und API-Gateway zu realisieren. Beide Dienste benötigen keine Datenbankanbindung, sondern im Falle des Analyseservices einen Layer zum Durchführen der Berechnungen und im Falle des Gateways entsprechende HTTP-Clients, um die Kommunikation zwischen allen Services zu koordinieren.

Nachdem die Kommunikation mithilfe des Postman-Tools [34] getestet wurde, wird die Frontendanwendung entwickelt.

Dieses Kapitel beschreibt die besonders interessanten Aspekte der Implementierung.

6.1 Datenbankanbindung

6.1.1 SQLAlchemy Model

Um das zuvor erstellte Datenbankmodell in die Anwendung zu integrieren, werden zunächst SQLAlchemy-Modelle (siehe Abbildung 7) erzeugt. Jedes Modell repräsentiert dabei eine Datenbanktabelle, wobei die Attribute die jeweiligen Spalten innerhalb der Tabelle definieren. Beim erstmaligen Starten der Anwendung erzeugt SQLAlchemy die entsprechenden Tabellen und Spalten in der Datenbank automatisch.

```
class Question(Base):
    __tablename__ = "questions"
    id = Column(UUID(as_uuid=True), primary_key=True, unique=True, nullable=False, default=uuid.uuid4)
    survey_id = Column(UUID(as_uuid=True), ForeignKey("surveys.id"))
    order = Column(Integer)
    question_text = Column(String)
    type = Column(Integer, nullable=False)
    options = Column(ARRAY(String))
    survey = relationship("Survey", back_populates="questions")
    responses = relationship("Response", back_populates="question", cascade="all, delete-orphan")
```

Abbildung 7: SQLAlchemy Data Model

Zum Generieren einer eindeutigen ID wird der Datentyp UUID Version 4 verwendet, da dieser allgemein als kollisionssicher und einzigartig gilt. Mithilfe der Eigenschaften "ForeignKey", "relationship" und "cascade" werden Beziehungen zwischen verschiedenen Datenbanktabellen hergestellt und die Datensätze miteinander verknüpft. In diesem Zusammenhang dient die Spalte "survey_id" in der "questions"-Tabelle als Fremdschlüssel (ForeignKey), der auf den Primärschlüssel "id" in der "surveys"-Tabelle verweist.

Beim Hinzufügen eines neuen Eintrags in die Datenbank für eine Antwort (Response) wird automatisch der Eintrag der zugehörigen Fragestellung ermittelt und zur Liste "responses" hinzugefügt. Dies geschieht durch die Verwendung der "relationship"-Eigenschaft und "back-populates", wodurch keine manuelle Pflege des zugehörigen Question-Eintrags mehr erforderlich ist.

Darüber hinaus bietet die Eigenschaft "cascade='all, delete-orphan'" die Möglichkeit, beim Löschen eines Question-Eintrags aus der Datenbank alle zugehörigen Response-Einträge mit dem entsprechenden Fremdschlüssel automatisch zu löschen.

Diese Funktionalitäten erweisen sich als äußerst nützlich, da sie die Integrität der persistierten Daten gewährleisten, potenzielle Fehlerquellen minimieren und damit die Wartbarkeit der Anwendung verbessern.

6.1.2 Pydantic Schemas

Im nächsten Schritt werden Pydantic-Schemas erzeugt (siehe Abbildung 8). Sie definieren die Struktur der von den Endpunkten zu verarbeitenden Daten und bilden damit die Grundlage zur Datenvalidierung und Serialisierung.

```
def convert_survey_model_to_schema(model: SurveyModel) -> SurveySchema:
    return SurveySchema(
        id=str(model.id),
        creator_id=str(model.creator_id),
        title=model.title,
        description=model.description,
        is_public=model.is_public,
        questions=[convert_question_model_to_schema(question) for question in model.questions]
    )
```

Abbildung 8: Methode zur Umwandlung von Models in Schemas

Da SQLAlchemy die aus der Datenbank abgerufenen Daten als Modelle zurückgibt (siehe Abbildung 7), ist es erforderlich, diese manuell in entsprechende Pydantic-Schemas umzuwandeln. Zu diesem Zweck ist zwischen API- und Datenbankschicht eine Serviceschicht implementiert. Diese stellt die Daten als Pydantic-Schemas zur Verfügung, indem sie entsprechende Umwandlungsmethoden verwendet (siehe Abbildung 8).

```
class UserCreate(BaseModel):
    name: str
    password: str
    role: str
    token: Optional[str] = None

class User(UserCreate):
    id: UUID4

class UserResponse(BaseModel):
    id: UUID4
    name: str
    role: str
    token: Optional[str]

    @classmethod
    def from_user(cls, user: User):
        return cls(
            id=user.id,
            name=user.name,
            role=user.role,
            token=user.token
        )
```

Abbildung 9: Pydantic User Schema

Die Aufteilung der Nutzerklassen in „UserCreate“, „User“ und „UserResponse“ (siehe Abbildung 9) legt fest, welche Daten in den verschiedenen Teilen der Anwendung verarbeitet und freigegeben werden.

So definiert „UserCreate“, welche Daten zum Erstellen eines neuen Nutzerprofils notwendig sind und vom Client gefordert werden. „User“ erweitert „UserCreate“ um die „id“, die nicht vom Client definiert, sondern von der Datenbank automatisch generiert wird. Dies ermöglicht eine nahtlose Integration in die Datenbank.

„UserResponse“ hingegen bietet mithilfe der „from_user“-Methode Kontrolle darüber, welche Daten nach außen freigegeben werden um so sensible Informationen wie beispielsweise das Passwort (in diesem Fall gehashed) zu schützen.

Insgesamt ermöglicht diese Aufteilung eine präzise Steuerung der Daten, und erhöht sowohl Sicherheit als auch Wartbarkeit der Anwendung.

6.2 HTTP APIs

API-Endpunkte sind ein wichtiger Bestandteil einer FastAPI-Anwendung und ermöglichen die Kommunikation zwischen mehreren Services via HTTP. Sie bilden die Schnittstelle, über die mit einem Service kommuniziert wird.

6.2.1 Endpunktdefinition

```
@router.post(
    "/questions/",
    response_model=Question,
    responses={
        400: {
            "description": "Integrity Error",
            "content": {
                "application/json": {
                    "example": {
                        "detail": "Can not create Question. No Survey with ID {uuid} found"
                    }
                }
            }
        }
    }
)
async def create_question(question: QuestionCreate, db: Session = Depends(get_db)):
    return await question_service.create_question(db, question)
```

Abbildung 10 Umfrageverwaltung create_question Endpunkt

```
async def create_question(question: QuestionCreate):
    url = "http://localhost:8000/questions/"
    ...
    async with httpx.AsyncClient() as client:
        response = await client.post(url, data=json_question)
    ...
```

Abbildung 11: API-Gateway create_question Client

Die Annotation „@router.post“ (siehe Abbildung 10) kennzeichnet einen POST-Endpunkt innerhalb einer FastAPI-Anwendung. Das erste übergebene Argument ist der Pfadparameter, der den Pfad definiert, über welchen dieser Endpunkt von anderen Services erreicht wird. In diesem Beispiel (siehe Abbildung 11) kann das API-Gateway Anfragen an „http://<hostname>:<port>/questions/“ senden, mit dem Ziel die Methode „create_question“ unterhalb der Annotation auszuführen.

6.2.2 Datenvalidierung

Das „response_model“ (siehe Abbildung 10) definiert das Pydantic Schema „Question“ und erfüllt dabei mehrere Aufgaben: Einerseits sorgt es für eine implizite Datenvalidierung, indem FastAPI vor der Antwortübermittlung die Übereinstimmung der herauszugebenen Daten mit dem Schema kontrolliert. Andererseits wird das „response_model“ in der von FastAPI automatisch generierten API-Dokumentation verwendet und dient damit dem Entwickler als Referenz, in welchem Format eine Antwort von diesem Endpunkt zu erwarten ist.

Da es sich in diesem Fall um einen POST-Endpunkt handelt, erwartet die API eine entsprechende Payload mit den Informationen des Objektes, welches in diesem Fall in der Datenbank persistiert werden soll. Um sicherzustellen, dass die Daten im entsprechenden Format gesendet und damit erfolgreich deserialisiert werden können, erhält die Methode als erstes Argument das Pydantic-Schema „QuestionCreate“. Somit kann FastAPI, wie schon beim „response_model“, direkt am Einstiegspunkt die Korrektheit der Daten validieren und das entsprechende Schema der automatisch generierten API-Dokumentation beifügen.

6.2.3 Exception Handling

```
async def create_question(db: Session, question: schemas.QuestionCreate):
    try:
        return await question_repository.create_question(db, question)
    except IntegrityError:
        raise HTTPException(
            status_code=400,
            detail=f"Can not create Question. No Survey with ID {str(question.survey_id)} found"
        )
```

Abbildung 12: Umfrageverwaltung HTTPException

FastAPI verfügt über ein integriertes Exception-Handling. Wann immer innerhalb der Anwendung eine HTTPException (siehe Abbildung 12) geworfen wird (in diesem Beispiel, falls versucht wird, eine Fragestellung in der Datenbank zu persistieren, ohne dass bereits eine zugehörige Umfrage existiert), wird diese Fehlermeldung direkt vom API-Endpunkt aufgefangen und als HTTP-Response mit dem entsprechenden Status-Code an den Client zurückgesendet. Um diese Fehlerfälle ebenfalls der API-Dokumentation hinzuzufügen, werden diese in der POST-Annotation unter „responses“ definiert (siehe Abbildung 10).

6.2.4 API-Dokumentation

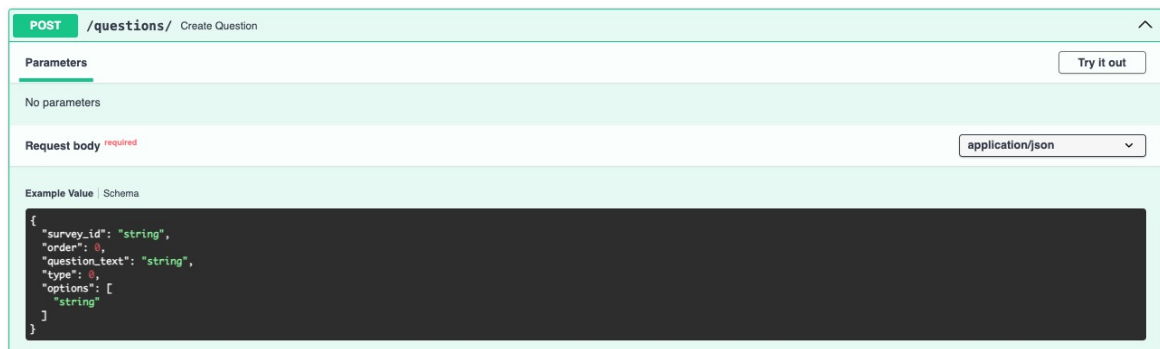


Abbildung 13: Auszug API-Dokumentation

Die von FastAPI automatisch generierte API-Dokumentation (siehe Abbildung 13) ist nach dem Start der Anwendung unter „{Pfad}/docs“ oder in alternativer Version unter „{Pfad}/redoc“ im Browser erreichbar. Sie enthält alle eben beschriebenen Informationen und ist bei Bedarf noch deutlich detaillierter definierbar.

6.2.5 Dependency Injection

```
SQLALCHEMY_DATABASE_URL = "postgresql://surveyuser:admin@localhost/surveydb"

engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Abbildung 14: Dependency Injection: Datenbankverbindung

Dependency-Injection wird verwendet, um dem Endpunkt Abhängigkeiten, in diesem Beispiel die Datenbankverbindung durch die FastAPI-Funktion „Depends“ (siehe Abbildung 10), zu übergeben. Die Logik der Methode „get_db“ (siehe Abbildung 14), eine neue Datenbankverbindung zu erstellen und nach erfolgreicher Abarbeitung wieder zu schließen, muss in diesem Fall nicht für jeden Endpunkt neu implementiert werden. Das verbessert die Wartbarkeit der Anwendung, da diese Funktionalität beliebig oft wiederverwendet werden kann. Darüber hinaus werden die einzelnen Komponenten voneinander entkoppelt und sind so unabhängig voneinander testbar und austauschbar.

6.3 Nutzerverwaltung

Die Nutzerverwaltung ermöglicht die Registrierung und Authentifizierung von Anwendern und stellt sicher, dass nur autorisierte Nutzer auf bestimmte Funktionalitäten zugreifen können. Das folgende Kapitel erläutert, wie die Nutzerverwaltung implementiert wurde.

6.3.1 Registrierung

```
@router.post("/users/register/")
async def register_user(user: user_schema.UserCreate, db: Session = Depends(get_db)):
    await user_service.register_user(db, user)
    return JSONResponse(content="User created successfully", status_code=200)

async def register_user(db: Session, user: UserCreate):
    user.password = __hash_password(user.password)
    await user_repository.create_user(db, user)

def __hash_password(password: str) -> str:
    return hashlib.sha256(password.encode('utf-8')).hexdigest()

async def create_user(db: Session, user: user_schema.UserCreate):
    existing_user = await get_user_by_name(db, user.name)
    if existing_user:
        raise HTTPException(
            status_code=400,
            detail="User already exists",
        )
    db_user = User(**dict(user))
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
```

Abbildung 15: Registrierung

Die Registrierung eines neuen Anwenders erfolgt über einen dedizierten "register"-Endpunkt in der API (siehe Abbildung 15). Dabei können Anwender ihren Namen, Passwort und Rollen entsprechend dem zuvor beschriebenen Pydantic-Schema "UserCreate" (siehe Abbildung 9) angeben. Nach erfolgreicher Übermittlung aller erforderlichen Daten erfolgt zunächst eine Hash-Transformation mithilfe des SHA-256-Algorithmus. Auf diese Weise wird sichergestellt, dass kein Passwort im Klartext in der Datenbank gespeichert wird. Anschließend wird eine eindeutige Nutzer-ID generiert und die Daten in der Datenbank gespeichert.

6.3.2 Anmeldung

Zur Anmeldung in der Anwendung ist ein "login_user"-Endpunkt (siehe Abbildung 16) implementiert, der Name und Passwort entgegennimmt.

```
@router.post("/users/login/")
async def login_user(login_data: user_schema.UserLogin, db: Session = Depends(get_db)):
    user = await user_service.authenticate_user(db, login_data)
    token = user_service.generate_token()
    await user_service.update_user_token(db, user.id, token)
    return await user_service.get_user(db, token)

async def authenticate_user(db: Session, login_data: UserLogin) -> User:
    user = await user_repository.get_user_by_name(db, login_data.name)
    if user and __check_password(login_data.password, user.password):
        return user
    raise HTTPException(status_code=401, detail="Invalid credentials")

def generate_token() -> str:
    return secrets.token_urlsafe(32)
```

Abbildung 16: Login

Die Nutzerverwaltung gleicht die übermittelten Daten mit denen der Datenbank ab und generiert im Erfolgsfall ein zufälliges Token. Dieses Token wird sowohl dem Nutzerobjekt in der Datenbank zugeordnet sowie als Teil der Antwort des „login_user“-Endpunktes an das Frontend zurückübermittelt. Da die Anwendung lediglich Demonstrationszwecken dient, wird hier auf ein sonst übliches Ablaufdatum des Tokens verzichtet. Das erhaltene Token wird nun im Vuex Store der Frontends gespeichert. Der Vuex Store ist ein State-Management-Tool in Vue.js und ermöglicht den Zugriff auf das Token von verschiedenen Teilen der Anwendung aus.

```
const token = this.$store.state.userToken;

export const postSurvey = async (token, survey) => {
    ...
    const response = await axios.post(
        `${BASE_URL}${SURVEYS_ENDPOINT}/`,
        survey,
        {
            headers: {
                'Authorization': `Bearer ${token}`
            }
        }
    );
    ...
}
```

Abbildung 17: API-Call mit Token

Führt der Nutzer nun innerhalb der Anwendung Aktionen, wie beispielsweise das Erstellen einer neuen Umfrage, aus, so wird das Token aus dem Vuex-Store als Header dem HTTP-Requests hinzugefügt (siehe Abbildung 17).

6.3.3 Autorisierung

Die Autorisierung von Nutzern erfolgt im API-Gateway, das als Vermittlungsinstanz zwischen Frontend und den Backendservices agiert.

```
@router.post("/surveys/", response_model=Survey)
@has_role(["editor"])
async def create_survey(
    request: Request,
    survey: SurveyCreate,
    user_id: str = Depends(get_current_user_id)
):
    return await survey_service_client.create_survey(survey, user_id)
```

Abbildung 18: API-Gateway `create_survey` Endpunkt

Erhält das Gateway einen API-Request, da der Anwender beispielsweise eine Umfrage erstellen und in der Datenbank persistieren möchte, ist zunächst zu prüfen, ob er über die notwendigen Berechtigungen verfügt. Dazu wird zunächst das Token aus dem Header extrahiert und eine Anfrage an die Nutzerverwaltung gesendet, um das entsprechende Objekt mit dem angegebenen Token aus der Datenbank abzurufen. Im Erfolgsfall wird das Nutzerobjekt als `UserResponse`-Objekt (siehe Abbildung 9) an das Gateway zurück gesendet. Dieses prüft nun die anhand der Rolle, ob ein Nutzer über die geforderten Berechtigungen verfügt und leitet die Anfrage im Anschluss an die entsprechenden Backendservices weiter.

```
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

def has_role(allowed_roles: list):
    def decorator(func):
        @wraps(func)
        async def wrapper(request: Request, *args, **kwargs):
            token = await oauth2_scheme.__call__(request)
            if not token:
                raise HTTPException(
                    status_code=401,
                    detail="Unauthorized",
                )
            user = await user_service_client.fetch_user_by_token(token)

            if user.role not in allowed_roles:
                raise HTTPException(
                    status_code=403,
                    detail="Forbidden",
                )
            return await func(request, *args, **kwargs)

        return wrapper

    return decorator

async def get_current_user_id(token: str = Depends(oauth2_scheme)):
    user = await user_service_client.fetch_user_by_token(token)
    return user.id if user else None
```

Abbildung 19: Dekorator `hasRole`

Um den Code übersichtlicher und einfacher wartbar zu gestalten, wurde in der Anwendung ein spezieller Dekorator "hasRole" (siehe Abbildung 19) implementiert. Dieser Dekorator wird den Controller-Methoden (siehe Abbildung 18) im API-Gateway hinzugefügt und erhält eine Liste von Rollen, die dazu berechtigt sind, die jeweilige Methode auszuführen.

Der Dekorator extrahiert bei einer eintreffenden Anfrage das Token und übernimmt anschließend die eben beschriebene Kommunikation mit der Nutzerverwaltung. Insofern die Rolle des Nutzers in der übergebenen Argumentliste vorhanden ist, wird die Anfrage ausgeführt. Zusätzlich erlaubt die „get_current_user_id“-Methode das Extrahieren der „id“ aus dem erhaltenen „UserResponse“ Objekt und übergibt sie via Dependency Injection als Argument an die Methode des aufgerufenen API-Endpunktes.

6.4 Analyseservice

Der Analyseservice ist insofern unabhängig vom Rest der Anwendung implementiert, dass dieser als Input lediglich ein Pydantic Schema der Fragestellung erhält, und dieses im Output um 2 optionale Felder (siehe Abbildung 20) mit den Ergebnissen der durchgeführten Analyse erweitert.

```
class Question(BaseModel):
    id: UUID4
    survey_id: UUID4
    order: int
    question_text: str
    type: int
    options: Optional[List[str]] = None
    responses: list[Response] = []

class AnalyzedQuestion(Question):
    analysis_responses: Optional[dict] = None
    analysis_respondents: Optional[dict] = None
```

Abbildung 20: AnalyzedQuestion

Als Datentyp wurde je ein Dictionary mit der Intention gewählt, die Analyseergebnisse möglichst generisch und damit erweiterbar zu halten, da diese intern um weitere Schlüssel-Wert-Paare erweitert werden können, und vom Client bei Bedarf ausgelesen werden können. Das neue „AnalyzedQuestion“-Objekt muss nun lediglich dem entsprechenden Client (hier API-Gateway) als erwartete Antwort hinzugefügt werden. Eine Anpassung in der übrigen Anwendung ist nicht erforderlich.

7 Tests

Tests dienen zur Prüfung der gegenwärtigen Funktionalität und gleichzeitigen Sicherstellung, dass nach künftigen Änderungen am Code, die gewünschte Funktionsweise weiterhin besteht. Zum Testen der Anwendung wurden dazu beispielhaft verschiedene Testszenarien mit Hilfe des Testframeworks „unittest“, dem eigenen TestClient von FastAPI und weiteren Werkzeugen implementiert. Im Folgenden wird die Umsetzung von Unit- und Integrationstests anhand kurzer Beispiele aufgezeigt.

7.1 Unittest

Unittests haben die Aufgabe die Funktionalität von Methoden innerhalb einer Komponente zu testen und sollen dabei möglichst jedes Ausgangsszenario abdecken. Die gewünschte Funktionalität der Anwendung wird so bereits auf kleinster Ebene sichergestellt.

```
#get_survey Methode als Referenz
async def get_survey(db: Session, survey_id: UUID):
    db_survey = await survey_repository.get_survey(db, survey_id)
    if db_survey is None:
        raise HTTPException(
            status_code=404,
            detail=f"Survey with ID {str(survey_id)} not found"
        )
    return convert_survey_model_to_schema(db_survey)
```

Abbildung 21: *get_survey Methode*

Die Methode „get_survey“ (siehe Abbildung 21) wird vom API-Endpunkt nach Aufforderung durch einen Client aufgerufen und ist dafür verantwortlich, das Survey-Repository mit einem Datenbankaufruf zu beauftragen. Wird das entsprechende Objekt in der Datenbank gefunden, soll dieses im Ergebnis an den API-Endpunkt zurückgegeben werden. Falls kein passendes Objekt gefunden wurde, wird eine HTTPException mit der entsprechenden Information, dass sich keine Umfrage mit der gesuchten „survey_id“ in der Datenbank befinden, geworfen. Diese Exception wird vom API-Endpunkt gefangen und eine Antwort mit Fehlercode 404 (Not Found) an den Client gesendet.

```

@patch("app.repository.survey_repository.get_survey")
def test_get_survey(self, mock_get_survey):
    survey_id = uuid4()

    test_survey = Survey(
        id=str(survey_id),
        creator_id=str(uuid4()),
        title="Survey Title",
        description="Survey Description",
        questions=[]
    )
    mock_get_survey.return_value = test_survey

    result = asyncio.run(get_survey(self.db, survey_id))

    self.assertEqual(test_survey, result)
    self.assertIsInstance(result, Survey)
    mock_get_survey.assert_called_once_with(self.db, survey_id)

```

Abbildung 22: Unittest positiv

Dieser Unittest (siehe Abbildung 22) simuliert zunächst durch die „@patch“ Annotation den Datenbankaufruf innerhalb des Survey-Repositorys: Sobald das Repository mit der spezifischen „survey_id“ aufgerufen wird, gibt dieses das zuvor erstellte „test_survey“-Objekt an den Aufrufer zurück.

Der eigentliche Test besteht nun darin, die „get_survey“-Methode im Survey Service mit der „survey_id“ aufzurufen und im Ergebnis zu prüfen, ob sie wie erwartet, das Repository aufgerufen und das Testobjekt erhalten hat. Zu diesem Zweck werden die „assert“-Methoden des „unittest“-Frameworks verwendet. Es wird geprüft, ob beide Objekte identisch sind, dass das zurückgegebene Objekt dem Pydantic Schema Survey entspricht, und ob das Repository mit den entsprechenden Argumenten aufgerufen wurde.

```

@patch("app.repository.survey_repository.get_survey", return_value=None)
def test_get_survey_not_found(self, mock_get_survey):
    survey_id = uuid4()

    with self.assertRaises(HTTPException):
        asyncio.run(get_survey(self.db, survey_id))
    mock_get_survey.assert_called_once_with(self.db, survey_id)

```

Abbildung 23: Unittest negativ

In diesem Test (siehe Abbildung 23) wird über die „@patch“-Annotation beim Aufruf des Repositorys ein Rückgabewert „None“ simuliert, um das Verhalten der „get_survey“-Methode zu testen, falls kein passendes Datenobjekt in der Datenbank gefunden wurde. Mit Hilfe der „assertRaises“-Methode wird überprüft, ob beim Aufruf der Testmethode die erwartete HTTPException geworfen wurde.

7.2 Integrationstest

Die Aufgabe von Integrationstests besteht in der Prüfung, ob Komponenten innerhalb einer Anwendung wie erwartet zusammenarbeiten. Sie bilden damit die nächsthöhere Testebene nach den Unittests.

```
def test_survey_integration(self):
    survey_create_data = {
        "creator_id": str(uuid4()),
        "title": "Integration Test Survey",
        "description": "Integration Test Description",
        "questions": [],
    }
    create_response = self.client.post("/surveys/", json=survey_create_data)
    assert create_response.status_code == 200
    created_survey = schemas.Survey(**create_response.json())
    ...
    delete_response = self.client.delete(f"/surveys/{created_survey.id}")
    assert delete_response.status_code == 200
    assert delete_response.json() == f"Survey with ID {str(created_survey.id)} deleted successfully"
    ...
    get_deleted_response = self.client.get(f"/surveys/{created_survey.id}")
    assert get_deleted_response.status_code == 404
    assert get_deleted_response.json() == {"detail": f"Survey with ID {str(created_survey.id)} not found"}
    ...
```

Abbildung 24: Integrationstest

Dieser Integrationstest (siehe Abbildung 24) startet mit einer Anfrage an den API-Endpunkt, um eine neue Umfrage zu erstellen. Hierbei ist zu beachten, dass keine Funktionalität der darunter liegenden Komponenten simuliert und außerdem eine reale Testdatenbank erstellt wird. Nach Erhalt der Anfrage wird die Antwort aus Sicht des Clients überprüft und anschließend eine Anfrage zur Löschung des eben erstellten Objektes gesendet. Auch diese Antwort wird anschließend auf Korrektheit überprüft.

Dieses Szenario lässt sich durch weitere Anfragen beliebig erweitern. Dementsprechend sollte eine erneute Löschanfrage des gleichen Objektes nun einen Fehler zurückgeben, da das Objekt bereits zuvor gelöscht wurde. Auf diese Art lassen sich verschiedene Szenarien, und somit das Zusammenspiel der einzelnen Komponenten, aus Sicht eines Clients testen.

8 Demonstration und Evaluierung

Dieses Kapitel demonstriert sowohl das Ergebnis der Implementierung und evaluiert, inwiefern funktionale und nicht-funktionale Anforderungen erfüllt wurden.

8.1 Demonstration

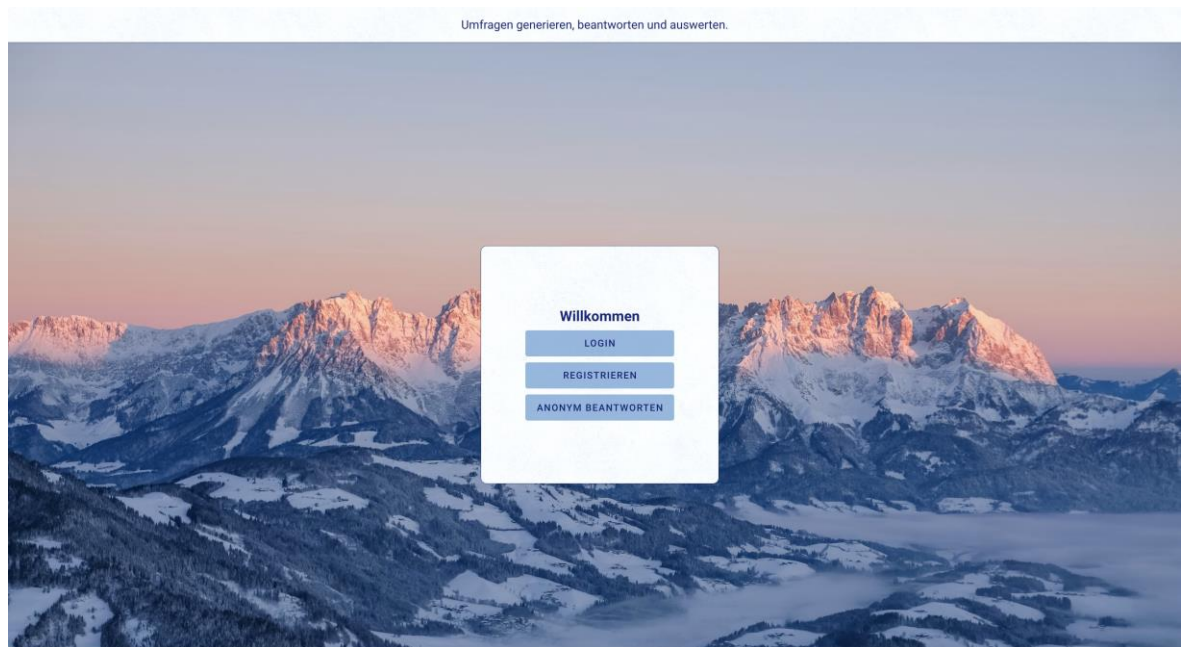


Abbildung 25: Landing Page

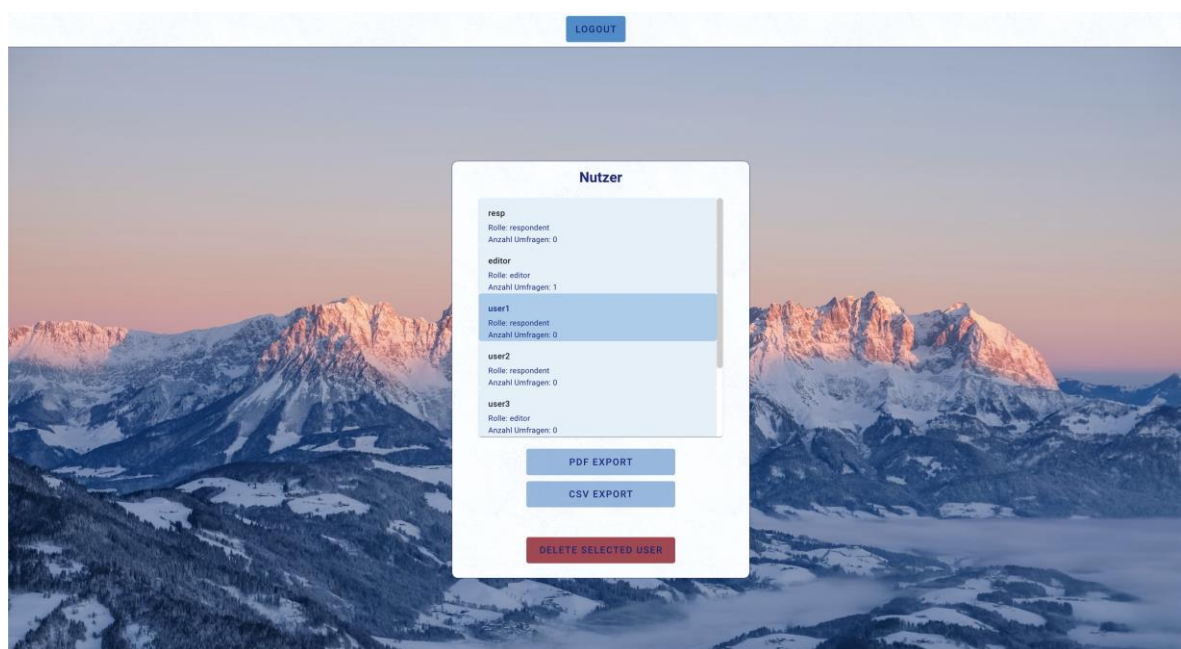


Abbildung 26: Admin Page

Die Landing Page (siehe Abbildung 25) bietet die Möglichkeit sich als neuer Nutzer zu registrieren, einzuloggen oder die öffentlichen Umfragen direkt anonym zu beantworten.

Die Admin Page (siehe Abbildung 26) ist für Nutzer mit der Rolle „admin“ erreichbar und enthält eine Liste der aktuell registrierten Nutzer inklusive zugehöriger Informationen. Der Export dieser Liste ist im PDF und CSV Format möglich. Ein selektierter Nutzer kann hier inklusive aller seiner erstellten Umfragen gelöscht werden, wobei ein Nutzer sich nicht selbst löschen kann.

Abbildung 27: Create-Survey-Page 2

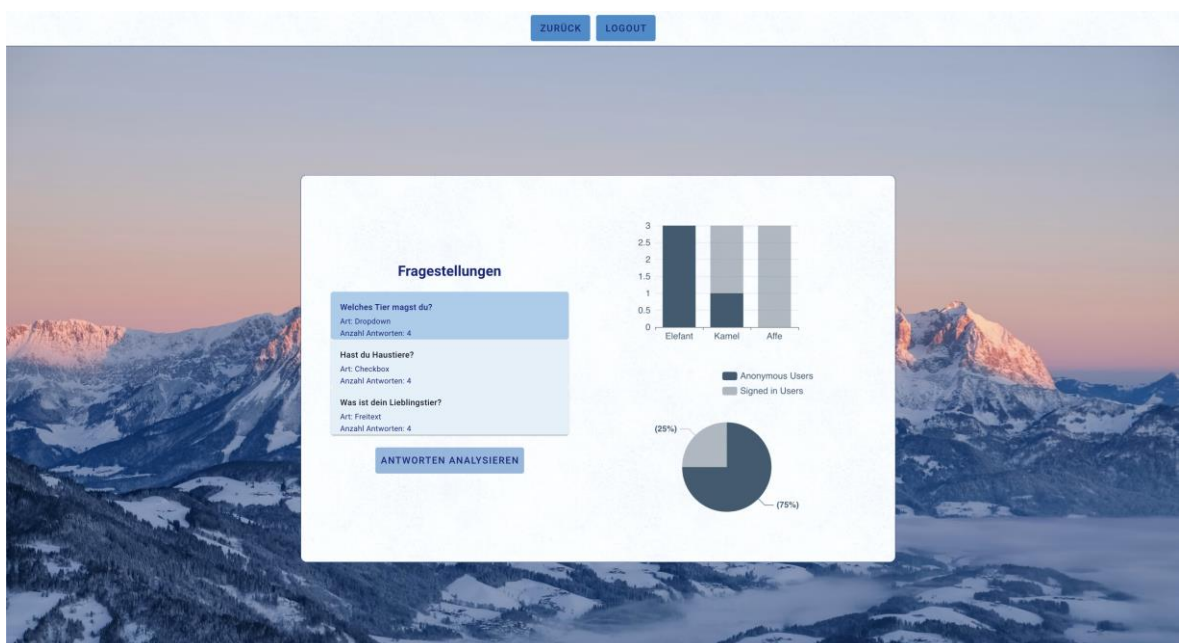


Abbildung 28: Analysis Page

Loggt sich ein Nutzer der Rolle „editor“ ein, so hat dieser nun die Möglichkeit eine Umfrage zu erstellen. Dazu müssen zunächst Titel, Beschreibung und Sichtbarkeit festgelegt werden um anschließend die Umfrage mit entsprechenden Fragestellungen zu füllen (siehe Abbildung 27). Jede neu erstellte Frage erscheint dazu rechts in einer Aufstellung, so dass der Editor jederzeit einen Überblick über die bereits erstellten Fragen behält und diese ggf. löschen kann.

Zudem hat der Nutzer Zugriff auf seine Umfragen, kann diese als PDF und CSV exportieren und sich Analysen der einzelnen Fragestellungen anzeigen lassen (siehe Abbildung 28)

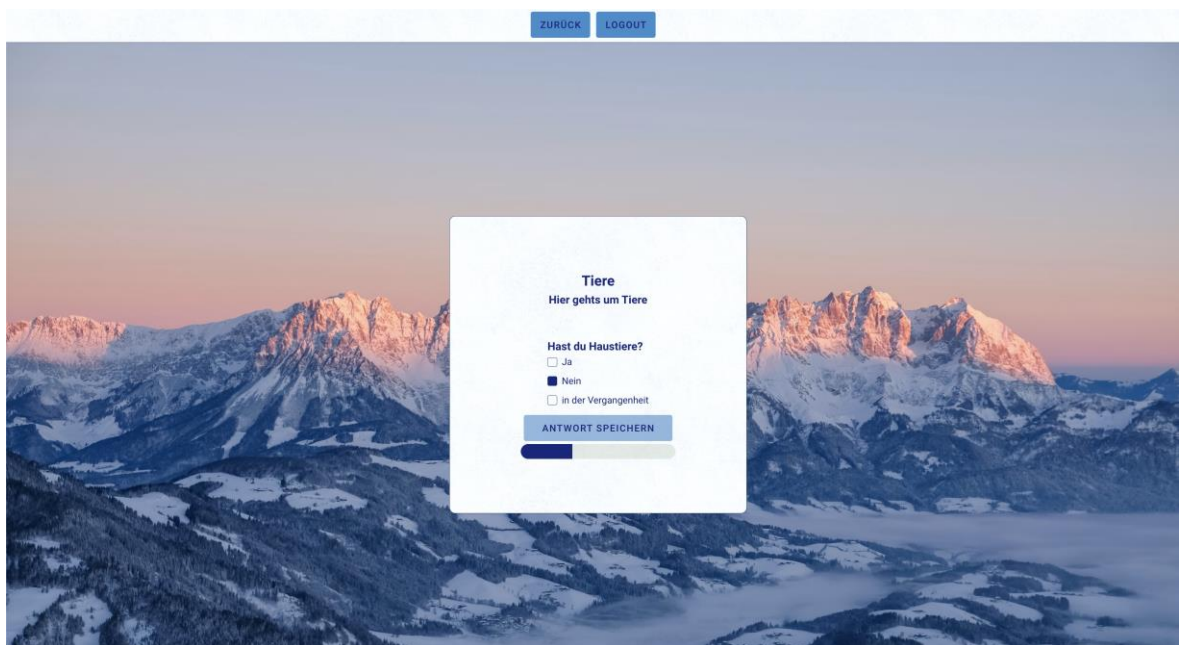
The image shows a web application interface for a survey. At the top, there are two buttons: 'ZURÜCK' and 'LOGOUT'. The background is a scenic image of snow-capped mountains at sunset. In the center, a white modal box contains the survey content. The title is 'Tiere' with the subtitle 'Hier gehts um Tiere'. The question is 'Hast du Haustiere?'. There are three radio button options: 'Ja' (unchecked), 'Nein' (checked), and 'in der Vergangenheit' (unchecked). Below the options is a blue button labeled 'ANTWORT SPEICHERN'. A progress bar is shown at the bottom of the modal, with the first segment filled in blue.

Abbildung 29: Respondent Page

Auf die Respondent Page gelangen sowohl eingeloggte Nutzer der Rolle „respondent“ sowie anonyme Nutzer direkt von der Startseite. Nachdem eine Umfrage aus der Liste der verfügbaren (anonyme Nutzer haben lediglich Zugriff auf öffentliche Umfragen) ausgewählt wurde, können hier nun die Fragen nacheinander beantwortet werden.

8.2 Evaluierung

FastAPI hat es ermöglicht, eine Anforderung zu implementieren, die sowohl den erwarteten funktionalen als auch nicht-funktionalen Anforderungen gerecht geworden ist. Die Entwicklung verlief zügig, unkompliziert und erforderte wenig Code. Alle Services konnten unabhängig voneinander entwickelt werden und teilen lediglich die erstellten Pydantic Schemas, die sich problemlos mit wachsenden Anforderungen erweitern lassen.

Die PostgreSQL Datenbank wurde mithilfe von SQLAlchemy unkompliziert angebunden und kann bei Bedarf durch alternative Lösungen ersetzt werden. Die integrierte Kaskadierungsfunktion der SQLAlchemy-Modelle ist dabei besonders hervorzuheben, da sie das Referenzieren und Löschen zusammengehöriger Datenobjekte übernimmt. Der verringerte Implementationsaufwand mindert die Anzahl potentieller Fehlerquellen und trägt damit maßgeblich zur Wartbarkeit und Robustheit bei.

Der modulare, in Microservices unterteilte Aufbau der Anwendung kann entsprechend der am Beginn dieser Arbeit beschriebenen wachsenden Anforderungen angepasst und erweitert werden. Dependency Injection ist hierbei ein hilfreiches Werkzeug, um Funktionalitäten wieder zu verwenden, den Code damit lesbarer zu gestalten und Verantwortlichkeiten einzelner Komponenten voneinander zu trennen.

Alle Services sind unabhängig voneinander implementiert und lassen sich mit Hilfe geeigneter Container Management Werkzeuge individuell skalieren. So ist es möglich auf steigende Nutzerzahlen ressourcenschonend zu reagieren, ohne die Leistungsfähigkeit der Gesamtwendung zu beeinträchtigen.

Das Testen der Endpunkte lässt durch den in FastAPI integrierten TestClient in Kombination mit dem Testframework „unittest“ keine Funktionalität vermissen. Somit wird sichergestellt, dass die Anwendung funktional den erwarteten Anforderungen entspricht.

Die automatisch generierte API-Dokumentation kann durch entsprechende Annotationen an den Endpunkten beliebig erweitert und den Anforderungen entsprechend angepasst werden.

Im Entwicklungsprozess war bei auftretenden Herausforderungen stets schnelle Unterstützung durch die offizielle Dokumentation, Online-Foren wie Stack Overflow und GitHub erreichbar.

Die nachfolgende Tabelle dient dazu, den Fertigstellungsgrad der funktionalen Anforderungen zu verdeutlichen.

Anforderung	Fertigstellungsgrad in %
Nutzerverwaltung	
Registrierung eines Nutzers	100
Authentifizierung eines Nutzers	100
Autorisierung entsprechend der Rolle des Nutzers	100
Umfrageverwaltung	
Erstellen von Umfragen	100
Beantworten von Umfragen	100
Löschen von Umfragen	100
Analyse	
Grafische Darstellung von Umfrageergebnissen	100
Datenexport	
CSV Export	100
PDF Export	100
GUI	
Grafische Nutzeroberfläche	100

Tabelle 1: Fertigstellungsgrad funktionale Anforderungen

8.3 Herausforderungen

Die größte Aufgabe bei der Umsetzung der Beispielanwendung besteht darin, einen geeigneten Systementwurf zu entwickeln, der sicherstellt, dass alle Anforderungen angemessen berücksichtigt werden. Dazu mussten im Implementierungsprozess geringfügige Anpassungen der Verantwortlichkeiten der einzelnen Services vorgenommen und Datenmodelle um zusätzliche Felder erweitert werden. Zur Datenbankmigration wurde das Werkzeug Alembic verwendet. Der Systementwurf wurde entsprechend iterativ angepasst, um das Verhalten der Anwendung zu jedem Zeitpunkt korrekt widerzuspiegeln.

Eine weitere Herausforderung liegt darin, einen geeigneten Kompromiss zwischen Funktionalität und Einfachheit zu finden. Ziel war es, ein System zu entwickeln, das die Einsatzmöglichkeiten von FastAPI aufzeigt, ohne durch ein übermäßiges Niveau an Komplexität den Rahmen dieser Arbeit zu überschreiten. Im folgenden Kapitel wird diesbezüglich ein Ausblick gegeben, welche Erweiterungen des Systems sinnvoll wären, um die Beispielanwendung in ein real einsetzbares System zu überführen.

9 Ausblick und Zusammenfassung

9.1 Ausblick

Dieser Absatz liefert verschiedene Ideen zur Verbesserung und Erweiterung der Beispielanwendung.

Die Integration externer Message oder Event Broker ermöglicht asynchrone beziehungsweise reaktive Kommunikation zwischen den Diensten. Komplexere Analysen könnten so asynchron durchgeführt werden. Zudem kann ein Benachrichtigungssystem mit dem Ziel Editoren regelmäßig über den Rücklauf zu bestimmten Umfragen zu informieren, implementiert werden.

Des Weiteren könnten Analyse- und Nutzerservice durch komplexere Versionen mit mehr Funktionalität und höherem Sicherheitsstandard ersetzt werden.

Auf technischer Ebene ist es vorstellbar, das Datenmodell feiner aufzugliedern und beispielsweise eigene Entitäten für die Frageoptionen zu entwerfen. Ebenso stellt ein Tracking innerhalb der Umfragen sicher, dass Nutzer die Umfragen je nur einmal ausfüllen können.

9.2 Zusammenfassung

Ziel dieser Bachelorarbeit ist es, ein Python Webframework zu ermitteln, dass sich gut zur Realisierung einer Softwareanwendung auf Grundlage einer Microservice-Architektur eignet. Aus mehreren Kandidaten wurden zunächst Django und FastAPI ausgewählt und anhand von Kriterien, die für Microservices von besonderer Bedeutung sind, miteinander verglichen. Beide Frameworks haben sich für diese Aufgabe als äußerst geeignet erwiesen, wobei FastAPI aufgrund seines speziellen Fokus auf der Realisierung von APIs hervorsticht und einen insgesamt moderneren Eindruck erweckt. Infolgedessen wurde FastAPI ausgewählt, um diese Erwartungen in der Praxis zu prüfen. Dazu wurde eine Beispielanwendung auf Grundlage einer Microservice-Architektur entworfen und anschließend implementiert.

Im Ergebnis hat sich die Arbeit mit FastAPI als effizient und vielversprechend erwiesen. FastAPI ist leicht zu erlernen, erfordert wenig Code und bewältigt die Anforderungen an ein auf Microservice basiertes System, wie Wartbarkeit und Skalierbarkeit, mühelos.

Die offizielle Dokumentation sowie der verfügbare Communitysupport bieten genügend Hilfestellung, um bei auftretenden Herausforderungen schnell entsprechende Lösungsansätze zu finden.

Abbildungsverzeichnis

Abbildung 1: Vergleich GitHub Stars [6].....	6
Abbildung 2: Vergleich Package Downloads Django und FastAPI [30]	15
Abbildung 3: Vergleich Google Trends Django und FastAPI [31]	16
Abbildung 4: Use Case Diagramme	23
Abbildung 5: Entwurf Systemarchitektur	24
Abbildung 6: UML-Klassendiagramm Data Model	27
Abbildung 7: SQLAlchemy Data Model.....	29
Abbildung 8: Methode zur Umwandlung von Models in Schemas.....	30
Abbildung 9: Pydantic User Schema	31
Abbildung 10 Umfrageverwaltung create_question Endpunkt.....	32
Abbildung 11: API-Gateway create_question Client.....	32
Abbildung 12: Umfrageverwaltung HTTPException.....	33
Abbildung 13: Auszug API-Dokumentation	34
Abbildung 14: Dependency Injection: Datenbankverbindung.....	34
Abbildung 15: Registrierung	35
Abbildung 16: Login.....	36
Abbildung 17: API-Call mit Token	36
Abbildung 18: API-Gateway create_survey Endpunkt.....	37
Abbildung 19: Dekorator hasRole.....	37
Abbildung 20: AnalyzedQuestion.....	38
Abbildung 21: get_survey Methode	39
Abbildung 22: Unittest positiv	40
Abbildung 23: Unittest negativ	40
Abbildung 24: Integrationstest	41
Abbildung 25: Landing Page	42

Abbildung 26: Admin Page	42
Abbildung 27: Create-Survey-Page 2	43
Abbildung 28: Analysis Page	43
Abbildung 29: Respondent Page	44

Abkürzungsverzeichnis

API	Application Programming Interface, Programmierschnittstellen
ASGI	Asynchronous Server Gateway Interface
CRUD	Create Read Update Delete
CSRF	Cross Site Request Forgery
CSV	Comma-separated values
FAQ	Frequently Asked Questions
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ID	Identifier
JSON	JavaScript Object Notation
MVC	Model View Controller
MVT	Model View Template
OAuth	Open Authorization
ORM	Object-relational mapping
PDF	Portable Document Format
REST	Representational State Transfer
SHA	Secure Hash Algorithm
SQL	Structured Query Language
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
WSGI	Web Server Gateway Interface

Glossar

Framework

„Ein Framework (englisch für Rahmenstruktur) ist ein Programmiergerüst, das in der Softwaretechnik, insbesondere bei der objektorientierten Softwareentwicklung sowie bei komponentenbasierten Entwicklungsansätzen, verwendet wird.“ [35]

Webframework

„Ein Webframework (...) ist ein Framework, das für die Entwicklung von dynamischen Webseiten, Webanwendungen oder Webservices ausgelegt ist. Sich wiederholende Tätigkeiten werden vereinfacht und die Wiederverwendung von Code (...) gefördert.“ [36]

Object-relational mapping

„Objektrelationale Abbildung (englisch object-relational mapping, ORM) ist eine Technik der Softwareentwicklung, mit der ein in einer objektorientierten Programmiersprache geschriebenes Anwendungsprogramm seine Objekte in einer relationalen Datenbank ablegen kann.“ [37]

Cross Site Request Forgery

„Eine Cross-Site-Request-Forgery (...) ist ein Angriff auf ein Computersystem, bei dem der Angreifer eine Transaktion in einer Webanwendung durchführt.“ [38]

Quellenverzeichnis

- [1] Atlassian, „Microservices vs. monolithic architecture“, Atlassian. Zugriffen: 11. Dezember 2023. [Online]. Verfügbar unter: <https://www.atlassian.com/de/microservices/microservices-architecture/microservices-vs-monolith>
- [2] „Python’s Frameworks Comparison: Django, Pyramid, Flask, Sanic, Tornado, BottlePy and More“. Zugriffen: 22. Januar 2024. [Online]. Verfügbar unter: <https://www.netguru.com/blog/python-frameworks-comparison>
- [3] D. Patel, „Build Microservices in Python: Business Guide“, Inexture. Zugriffen: 22. Januar 2024. [Online]. Verfügbar unter: <https://www.inexture.com/build-microservices-in-python/>
- [4] „Discover the Power of Python Microservices: Transforming Application Development for the Modern Age“. Zugriffen: 22. Januar 2024. [Online]. Verfügbar unter: <https://www.linkedin.com/pulse/discover-power-python-microservices-transforming-application-shibu-kt>
- [5] „Django vs. Pyramid - comparing Python Web Frameworks“, Sunscrapers. Zugriffen: 22. Januar 2024. [Online]. Verfügbar unter: <https://sunscrapers.com/blog/django-vs-pyramid-comparing-a-python-web-frameworks/>
- [6] „GitHub Star History“. Zugriffen: 14. Dezember 2023. [Online]. Verfügbar unter: <https://star-history.com/#django/django&pallets/flask&cherrypy/cherrypy&bottlepy/bottle&tiangolo/fastapi&Pylons/pyramid&tornadoweb/tornado&sanic-org/sanic&Date>
- [7] A. Holovaty und J. Kaplan-Moss, *The Definitive Guide to Django*. Berkeley, CA: Apress, 2008. doi: 10.1007/978-1-4302-0331-5.
- [8] „9 Examples of Companies Using Django in 2023 | Trio Developers“. Zugriffen: 16. Dezember 2023. [Online]. Verfügbar unter: <https://www.trio.dev/blog/django-applications>
- [9] N. Litzel und S. Luber, „Was ist Flask?“, BigData-Insider. Zugriffen: 17. Dezember 2023. [Online]. Verfügbar unter: <https://www.bigdata-insider.de/was-ist-flask-a-994166/>
- [10] „Flask: Alles Wichtige zum Micro-Framework“, IONOS Digital Guide. Zugriffen: 16. Dezember 2023. [Online]. Verfügbar unter: <https://www.ionos.de/digitalguide/websites/web-entwicklung/flask-framework-im-ueberblick/>
- [11] V. Podoba, „The Best Python Frameworks for Web Development“, SOFTFORMANCE. Zugriffen: 17. Dezember 2023. [Online]. Verfügbar unter: <https://www.softformance.com/blog/python-web-frameworks/>
- [12] A. R., „FastAPI: Alles über den meistgenutzten Python-Framework“, Weiterbildung Data Science | DataScientest.com. Zugriffen: 16. Dezember 2023. [Online]. Verfügbar unter: <https://datascientest.com/de/fastapi>
- [13] „History, Design and Future - FastAPI“. Zugriffen: 16. Dezember 2023.

- [Online]. Verfügbar unter: <https://fastapi.tiangolo.com/history-design-future/>
- [14] „Alternatives, Inspiration and Comparisons - FastAPI“. Zugegriffen: 16. Dezember 2023. [Online]. Verfügbar unter: <https://fastapi.tiangolo.com/alternatives/>
- [15] „What is Container Load Balancing? Definition & FAQs“, Avi Networks. Zugegriffen: 21. Dezember 2023. [Online]. Verfügbar unter: <https://avinetworks.wpengine.com/glossary/container-load-balancing/>
- [16] A. Jaffery, „Asynchrone API: Der Schlüssel zu Skalierbarkeit und Leistung“, Astera. Zugegriffen: 21. Dezember 2023. [Online]. Verfügbar unter: <https://www.astera.com/de/topic/api-management/asynchronous-api-the-key-to-scalability-and-performance/>
- [17] „Web Server Gateway Interface“, *Wikipedia*. 10. Dezember 2023. Zugegriffen: 22. Dezember 2023. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=Web_Server_Gateway_Interface&oldid=240054389
- [18] „Introduction — WSGI Tutorial“. Zugegriffen: 22. Dezember 2023. [Online]. Verfügbar unter: <https://wsgi.tutorial.codepoint.net/intro>
- [19] „Asynchronous Server Gateway Interface“, *Wikipedia*. 13. Dezember 2023. Zugegriffen: 22. Dezember 2023. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=Asynchronous_Server_Gateway_Interface&oldid=1189756489
- [20] „Django“, Django Project. Zugegriffen: 22. Dezember 2023. [Online]. Verfügbar unter: <https://docs.djangoproject.com/en/5.0/howto/deployment/wsgi/>
- [21] A. I. Bharata, „Django Async vs FastAPI vs WSGI Django: Choice of ML/DL Inference Servers — Answering some burning...“, Medium. Zugegriffen: 22. Dezember 2023. [Online]. Verfügbar unter: <https://ai.plainenglish.io/django-async-vs-fastapi-vs-wsgi-django-choice-of-ml-dl-inference-servers-answering-some-burning-e6a354bf272a>
- [22] A. N. Oyom, „Understanding the MVC pattern in Django“, Nur: The She Code Africa Blog. Zugegriffen: 23. Dezember 2023. [Online]. Verfügbar unter: <https://medium.com/shocodeafrica/understanding-the-mvc-pattern-in-django-edda05b9f43f>
- [23] „Django“, Django Project. Zugegriffen: 23. Dezember 2023. [Online]. Verfügbar unter: <https://docs.djangoproject.com/en/4.2/faq/models/>
- [24] „Django“, Django Project. Zugegriffen: 24. Dezember 2023. [Online]. Verfügbar unter: <https://docs.djangoproject.com/en/5.0/topics/testing/>
- [25] „unittest — Unit testing framework“, Python documentation. Zugegriffen: 24. Dezember 2023. [Online]. Verfügbar unter: <https://docs.python.org/3/library/unittest.html>
- [26] „Django“, Django Project. Zugegriffen: 24. Dezember 2023. [Online]. Verfügbar unter: <https://docs.djangoproject.com/en/5.0/topics/testing/tools/>
- [27] „Test Client - Starlette“. Zugegriffen: 24. Dezember 2023. [Online]. Verfügbar unter: <https://www.starlette.io/testclient/>
- [28] „Testing - FastAPI“. Zugegriffen: 24. Dezember 2023. [Online]. Verfügbar

unter: <https://fastapi.tiangolo.com/tutorial/testing/>

[29] „spring-projects/spring-boot“. Spring, 17. Dezember 2023. Zugegriffen: 17. Dezember 2023. [Online]. Verfügbar unter: <https://github.com/spring-projects/spring-boot>

[30] „Django vs fastapi“, pip Trends. Zugegriffen: 17. Dezember 2023. [Online]. Verfügbar unter: <https://piptrends.com/compare/Django-vs-fastapi>

[31] „Google Trends“, Google Trends. Zugegriffen: 17. Dezember 2023. [Online]. Verfügbar unter: <https://trends.google.com/trends/explore?date=2018-12-01%202023-12-17&q=fastApi,python%20django&hl=de>

[32] „Django“, Django Project. Zugegriffen: 20. Dezember 2023. [Online]. Verfügbar unter: <https://docs.djangoproject.com/en/5.0/>

[33] „FastAPI“. Zugegriffen: 20. Dezember 2023. [Online]. Verfügbar unter: <https://fastapi.tiangolo.com/>

[34] „What is Postman? Postman API Platform“, Postman API Platform. Zugegriffen: 4. Februar 2024. [Online]. Verfügbar unter: <https://www.postman.com/product/what-is-postman/>

[35] „Framework“, *Wikipedia*. 5. Dezember 2023. Zugegriffen: 27. Januar 2024. [Online]. Verfügbar unter: <https://de.wikipedia.org/w/index.php?title=Framework&oldid=239844748>

[36] „Webframework“, *Wikipedia*. 17. Juli 2022. Zugegriffen: 27. Januar 2024. [Online]. Verfügbar unter: <https://de.wikipedia.org/w/index.php?title=Webframework&oldid=224573426>

[37] „Objektrelationale Abbildung“, *Wikipedia*. 19. Januar 2023. Zugegriffen: 27. Januar 2024. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=Objektrelationale_Abbildung&oldid=230015918

[38] „Cross-Site-Request-Forgery“, *Wikipedia*. 27. Juli 2023. Zugegriffen: 27. Januar 2024. [Online]. Verfügbar unter: <https://de.wikipedia.org/w/index.php?title=Cross-Site-Request-Forgery&oldid=235875073>

Tabellenverzeichnis

Tabelle 1: Fertigstellungsgrad funktionale Anforderungen.....	46
---	----