Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Bachelorarbeit

im Bachelor-Studiengang Wirtschaftsinformatik

Entwicklung einer Schnittstelle für die Anbindung von austauschbaren Datenquellen an KI-Algorithmen

von

Laurenz Anton Dilba

Erstprüfer:	Prof. Dr. Matthias Bertram
Zweitprüfer:	Prof. Dr. Wolfgang Heiden
Unternehmen:	CONET Solutions GmbH
Eingereicht am:	21. Dezember 2022

Erklärung

Hiermit erkläre ich wahrheitsgemäß, dass ich den vorliegenden Bericht selbst angefertigt habe. Der Bericht gibt die tatsächlich durchgeführten Arbeiten wieder. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Vertrauliche Informationen sind nicht enthalten.

Datum	Unterschrift Studierender	Unterschrift Betreuer

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AJAX Asynchronous JavaScript and XML
AMQP Advanced Message Queuing Protocol
API Application Programming Interface
BERT Bidirectional Encoder Representations from Transformers
BL Business Logic
DSGVO Datenschutz-Grundverordnung
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
ID Identifikation
IT Informationstechnik
JSON JavaScript Object Notation
JWT JSON Web Token
KI Künstliche Intelligenz
MAC Message Authentication Code
RAM Random-Access Memory
RDBMS relationalen Datenbankmanagementsystemen
REST Representational State Transfer
TCP Transmission Control Protocol
UI User Interface
URL Uniform Resource Locator
UUID Universally Unique Identifier
VM Virtuelle Maschine

1 Einleitung

text

1.1 Motivation und Hintergrund

text

1.2 Problemstellung

text

1.3 Zielsetzung

text

1.4 Stand der Forschung

text

1.5 Vorgehen

text

2 Grundlagen

2.1 Schnittstelle

text

2.2 REST und RabbitMQ

2.2.1 Application Programming Interface

Ein Application Programming Interface (API) ist eine Programmierschnittstelle, die dazu da ist, die Kommunikation zwischen einem Client, oder auch Anwender genannt, und einem Server durch festgelegte Funktionen zu regeln. Der Satz der verfügbaren Funktionen ist durch den Entwickler der API vorgegeben. Eine API sollte nach Möglichkeit selbsterklärend aufgebaut sein.[1] Eine API dient dazu, dem Nutzer Daten bereitzustellen oder dem Server Daten zu senden.

In Abbildung 1 ist die Kommunikation zwischen einem Anwender und der API durch ein Sequenzdiagramm dargestellt. Jede Anfrage an die API findet durch den Aufruf des API Endpunkts durch einen Uniform Resource Locator (URL) statt. Hinter der Grund-URL wird die genaue Ressource innerhalb der API durch den Pfad in der URL angegeben. Jede dieser Funktionen, die über eine bestimmte URL erreichbar sind, kann mehrere Effekte haben. Der Effekt, den die Funktion hat, ist durch die Hypertext Transfer Protocol (HTTP) Methoden im groben Rahmen vorgegeben. Die Kommunikation zwischen API und Anwender geht immer vom Anwender aus. Die API kann von sich aus den Anwender nicht ansprechen.

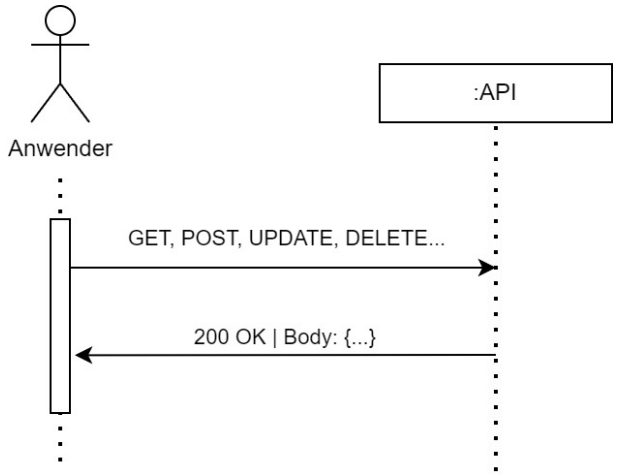

Abbildung 1: UML Sequenzdiagramm einer API

Der gesamte Satz der verfügbaren HTTP-Funktionen lautet `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `CONNECT`, `OPTIONS`, `TRACE`, `PATCH`. Jede der Funktionen hat eine bestimmte Aufgabe und bestimmte Rechte, die es vom Entwickler der API einzuhalten gilt.[2] In der Bachelorarbeit werden die Funktionen `GET`, `POST` und `DELETE` verwendet. Routen der API, die mit einer `GET` Funktion aufgerufen werden, sollen lediglich Daten an den Nutzer zurückgeben, ohne Änderungen am Server vorzunehmen. Wenn zwei mal hintereinander

[1] Bloch, 2006.
[2] MDN, 2022.

die gleiche URL mit `GET` aufgerufen wird, sollte auch beide male das gleiche Ergebnis von der API zurückgegeben werden.

Die `POST` Methode liefert Daten vom Nutzer an den Server. Innerhalb des `POST` Bodys können Daten unter Anderem im JSON Format an die API gesendet werden. API URLs die mit einer `POST` Methode aufgerufen werden, können Veränderungen auf dem Server auslösen. Dies kann Auswirkungen auf die Daten oder den Status des Servers haben, sodass eventuell andere Methoden davon beeinflusst sind. Ein beispielhafter Anwendungsfall für eine `POST` Methode ist die Registrierung eines neuen Accounts auf einer Website. Dort wird dann innerhalb des Bodys der Benutzername und das Passwort an die API geschickt. Die API Registriert den Account mit den erhaltenen Daten und gibt bei zukünftigen Aufrufen der URL mit den gleichen Daten einen Fehler zurück, dass der Benutzer bereits registriert ist.

Die `DELETE` Methode entfernt eine Ressource unter der aufgerufenen URL. Sowohl bei `POST`, als auch bei `DELETE` Methoden, ist darauf zu achten, dass der Nutzer die Route nur aufrufen kann, wenn er entsprechende Rechte zur Ausführung verfügt. Ansonsten könnte es zu unkontrollierten Daten- und Statusänderungen innerhalb der API kommen.

Wie in Abbildung 1 zu sehen ist, gibt die API nicht nur die angeforderten Daten zurück, sondern auch einen Statuscode. Ein HTTP Statuscode besteht aus drei Ziffern. In Tabelle 1 sind die grundlegenden Statuscodes aufgelistet.

Statuscode	**Bedeutung**
1xx	Informationen, nur unter HTTP 1.1 definiert
2xx	Request war erfolgreich (OK)
3xx	Die Ressource wurde verschoben
4xx	Die Eingabe war fehlerhaft
5xx	Der Server hatte einen Fehler

Tabelle 1: HTTP Statuscodes nach Doglio, 2015

Die Statuscodes, die im Prototypen der Bachelorarbeit verwendet wurden, sind Teil der häufiger genutzten Statuscodes für APIs. Der Statuscode `200:OK` ist der am häufigsten auftretende Statuscode. Dieser besagt, dass die Anfrage erfolgreich abgelaufen ist und das Ergebnis zurückgegeben werden konnte. Der Prototyp nutzt ein Authentifizierungssystem. Dadurch kann es dazu kommen, dass bei einer Anfrage mit fehlender Autorisierung der Statuscode `401:Unauthorized` zurückgegeben wird. Weitere Statuscodes, die häufiger auftreten können, sind `404:Not found`, `405:Method not allowed` und `500:Internal server error`. Bei einem Statuscode 404 wurde eine Route aufgerufen, die innerhalb der API nicht definiert ist. Bei dem Statuscode 405 wurde zwar eine vorhandene Route angesprochen, jedoch ist die genutzte HTTP Methode nicht zulässig. Der letzte verwendete Statuscode 500 beschreibt einen Serverabsturz bei der Ausführung der Anfrage.

Die HTTP Methoden und Statuscodes dienen dazu, die Entwicklung und Arbeit mit einer API für den Entwickler einfach zu gestalten. Wie die URLs der API aufgebaut sind und welche HTTP Methoden wann verwendet werden, liegt jedoch vollständig beim Entwickler der API. Um APIs im Allgemeinen etwas zu vereinheitlichen und die Effekte der

Funktionen innerhalb der API selbsterklärender werden zu lassen, wurde im Jahr 2000 von Roy Thomas Fielding ein Regelwerk namens Representational State Transfer (REST) veröffentlicht.[3]

2.2.2 Representational State Transfer

Der Representational State Transfer, kurz REST, ist kein spezifischer Standard in der Softwareentwicklung. REST ist eine Richtlinie, die den Aufbau der Kommunikation zu einer API vorgibt. APIs, die das REST-Paradigma implementieren, werden als REST-APIs oder RESTful APIs bezeichnet. Diese bieten damit einen größtenteils standardisierten Weg, Daten zwischen Client und Server auszutauschen.[4]

Innerhalb des REST-Frameworks sind mehrere Aspekte vorgegeben, die eine REST-API ausmachen. Drei der Aspekte sind „Simplizität“, „Skalierbarkeit“ und „Performance“. Die Simplizität beschreibt einen allgemeinen, standardisierten Weg, wie der Aufbau und die Kommunikation mit der API ablaufen soll. Dies beinhaltet den Aufbau der Routen und damit der URL, wann Daten an die API geschickt werden sollen, sowie die Form der Daten, die zu versenden sind.

Die Skalierbarkeit beschreibt das Konzept der beliebigen Erweiterung einer API. Das beinhaltet die Entwicklung der API als solche. Routen und damit Möglichkeiten, Daten von der API anzufordern oder Daten an die API zu senden, können während der Entwicklung einfach hinzugefügt und entfernt werden. Das Konzept der Skalierbarkeit beschreibt ebenfalls einen zustandslosen Aufbau der API. Durch die Zustandslosigkeit, kann die API horizontal skaliert werden. Eine horizontale Skalierung beschreibt das hinzufügen neuer Instanzen der API. Jede Instanz ist identisch aufgebaut und kann jede ankommende Anfrage alleinstehend beantworten. Die Anfragen, die an eine REST-API gestellt werden, müssen daher alle, für die Bearbeitung notwendigen, Informationen bereitstellen.

Der letzte Aspekt des REST-Frameworks beschreibt die Performance. Eine REST-API implementiert ein System zum Zwischenspeichern, auch Cachen gennant, von Responses. Wenn ein Client eine Anfrage mit der HTTP Methode GET an die API schickt, wird die Antwort, die an den Client zurückgesendet wird auf dem Server zwischengespeichert. Sollte ein oder mehrere Clients eine identische Anfrage an die API senden, wird das zwischengespeicherte Ergebnis zurückgegeben, statt die dahinter liegende Methode innerhalb der API neu auszuführen. Das ermöglicht eine hohe Performance, auch bei einer großen Anzahl von Anfragen.[5]

2.2.3 RabbitMQ

Kommunikation ist für den Aufbau von komplexen Strukturen essenziell. Das betrifft zum Beispiel die natürliche Sprache der Menschen, damit das Leben in einer Gesellschaft möglich wird. Für komplexe Programme in der Informationstechnik (IT) gelten die gleichen Prinzipien.[6]

[3] Masse, 2011.
[4] Richards, 2006.
[5] R.T. Fielding, 2000.
[6] Dossot, 2014.

Bei einer grundlegenden Kommunikation gibt es zwei Kommunikationspartner, einen Sender und einen Empfänger. Im Bereich der Software ist der Sender meist ein Client und der Empfänger ein Server, der öffentlich erreichbar ist. Der Client sendet eine Anfrage, der auch Request genannt wird, an den Server. Anschließend wartet der Client auf eine Antwort. Der Server erhält den Request und verarbeitet ihn. Es wird abhängig vom Request eine Antwort, die als Response bezeichnet wird, generiert und dem Client zurückgesendet. Der Client erhält die Response und schließt damit den Kommunikationsvorgang ab. Dieser Ablauf ist in Abbildung 1 visualisiert.

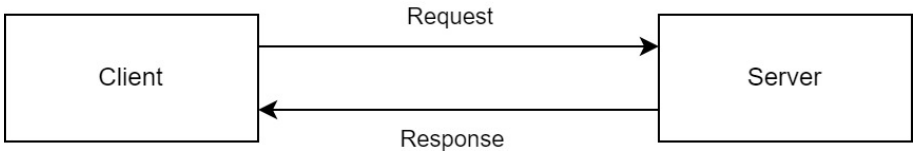

Abbildung 2: Grundlegende Kommunikation

Durch die Art der synchronen Kommunikation sind Client und Server sehr eng miteinander Verbunden. Wenn der Client erwartet eine Response von genau dem Server, an den er den Request geschickt hat. Das macht die Skalierung und Ausfallsicherheit allerdings relativ schwierig.

Damit eine Kommunikation zwischen unabhängigen Programmen möglich wird, muss es einen Zwischendienst geben, der die Nachrichten von einem Programm zum anderen transportiert. Bei der Kommunikation zwischen einer Website und einer API wird HTTP verwendet. Dieses stellt sicher, dass die die Nachrichten erfolgreich beim Empfänger ankommen. Sollte eine Nachricht nicht angekommen sein, hat der Absender die Möglichkeit, die Nachricht erneut zu schicken. Über HTTP wird automatisch eine erneute Anfrage geschickt, wenn keine Antwort vom Server zurück kam. Problematisch wird diese Herangehensweise, wenn die Antwortzeit sehr lang wird oder ungewiss ist, ob überhaupt eine Antwort kommen wird.

RabbitMQ ist ein eine nachrichtenorientierte Middleware, die eine Kommunikation zwischen zwei oder mehreren Programmen durch das Advanced Message Queuing Protocol (AMQP) ermöglicht. RabbitMQ dient als Broker, der Nachrichten von mehreren Clients an mehrere Server vermitteln kann. Im Gegensatz zu einer direkten Kommunikation zwischen Client und Server wie bei HTTP, wird in RabbitMQ eine Queue implementiert, in der alle Anfragen gesammelt werden.

Eine Queue ist eine Datenstruktur die mehrere Operationen definiert werden. Die Queue fungiert als Warteschlange, in der Einträge gespeichert und in der Reihenfolge des Eingangs auch wieder ausgelesen werden können. Sie funktioniert nach dem First In - First Out, kurz FIFO, Prinzip. Es wird eine `push` Operation definiert, mit der ein Eintrag an den Anfang der Warteschlange geschrieben wird. Des Weiteren gibt es eine `pop` Operation, die das älteste Element aus der FIFO Queue ausliest und es aus der Warteschlange entfernt.

Jeder Client kann Nachrichten in die Queue schreiben. Diese Nachrichten werden dort so lange gespeichert, bis sie von einem Dienst ausgelesen werden. In Abbildung 2 ist die Kommunikation zwischen Client und Server mittels eines Brokers abgebildet.

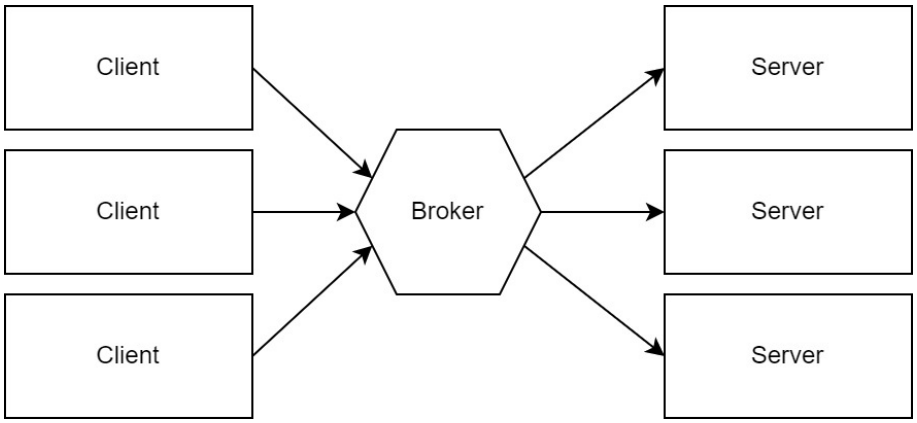

Abbildung 3: Kommunikation über einen Broker

Durch diese Herangehensweise wird eine asynchrone Kommunikation zwischen Client und Server ermöglicht. Da RabbitMQ frei von den Handshakes des HTTP ist, sind die Schreib- und Lesezeiten deutlich schneller.[7]

Die Middleware RabbitMQ wird für die Kommunikation zwischen der Flask API und den KI-Services genutzt. Der im Frontend vom Nutzer eingegebene Text-Input wird an die Flask API geschickt. Die Flask API modifiziert den Text im Anschluss so, dass es mittels der JavaScript Object Notation (JSON) über den RabbitMQ Service in die Queue geschrieben werden kann. Jeder KI-Service hat eine Queue einprogrammiert, aus der die Nachrichten ausgelesen werden. Diese Nachrichten können dann verarbeitet und im Anschluss in eine Response-Queue geschrieben werden. Das Flask Backend kann diese Response-Queue auslesen und die einzelnen Antworten zusammenbauen.

2.3 Künstliche Intelligenz

text[8]

2.3.1 Grundlagen einer KI

2.3.2 Künstliche Intelligenz als Service

Die KI-Services sind alleinstehende Programme, die die Aufgabe haben, Nachrichten anzunehmen, sie zu transformieren, zu analysieren und anschließend ein oder mehrere Ergebnisse zurückzugeben.

Um die Nachrichten empfangen und die Ergebnisse zurücksenden zu können, muss in jedem Service eine AMQP Verbindung zu RabbitMQ hergestellt werden.

Im Prototypen zur Anbindung von austauschbaren Datenquellen an KI-Algorithmen wurde ein Service zur Textähnlichkeitssuche implementiert. Dieser nimmt nutzt das Bidirectional Encoder Representations from Transformers (BERT) Modell von Google. Beim Starten des Services werden alle Einträge in einer Elasticsearch Datenbank mithilfe der künstlichen Intelligenz analysiert und in semantische Vektoren konvertiert. Der Service kann anschließend vom Nutzer eingegebene Anfragen mit dem gleichen BERT Modell ana-

[7] Ionescu, 2015.
[8] Görz u. a., 2010.

lysieren und den daraus entstandenen Vektor mit den Vektoren in der Datenbank abgleichen. Nach einem erfolgreichen Suchdurchlauf werden die semantisch ähnlichsten Einträge über RabbitMQ wieder an das Backend zurückgegeben.

2.4 Microservice Architekturen

text

2.5 Verwendete Werkzeuge

2.5.1 Python API mit Flask

Python ist eine um 1991 von Guido van Rossum entwickelte Programmiersprache. Bei der Entwicklung von Python wurde ein besonderer Fokus auf die Lesbarkeit von Code gesetzt. Dank der simplifizierten Syntax im Vergleich zu anderen höheren Programmiersprachen wie Java oder C#, ist Python auch in Bereichen, wie in der Mathematik oder der Wissenschaft ein häufig genutztes Werkzeug. Python bietet ebenfalls die Möglichkeit, von anderen Entwicklern bereitgestellte Bibliotheken in das eigene Projekt zu integrieren.[9]

Flask ist eine der verfügbaren Bibliotheken, die ein Framework für die Implementierung einer webbasierten API bereitstellt. Eine API dient dazu, Funktionen und Routen zu definieren, um die Kommunikation zwischen dem Frontend und dem Backend herzustellen. Das Flask Framework ist im Gegensatz zu anderen Frameworks sehr klein. Dies ermöglicht ein schnelles aufsetzen und entwickeln. Da Flask nur die nötigsten Grundlagen für eine API mitliefert, ist der Code besser lesbar und damit für andere Entwickler besser wartbar.[10]

Die Flask API wird für die Anbindung des Frontends an die Datenbank, sowie die Anbindung an die Kommunikationsschnittstelle von RabbitMQ verwendet. Sie nimmt die Daten oder die Eingaben des Nutzers entgegen und vermittelt sie an den richtigen Dienst, damit sie von einer KI-Schnittstelle ausgewertet werden können. Anschließend kann die API angefragt werden, ob es bereits Antworten von einer Künstliche Intelligenz (KI) zu der vorher geschickten Anfrage gab. Falls die API die Auswertung der KI erhalten hat, wird diese für das Frontend bereitgestellt, um sie dort anzeigen zu können.

2.5.2 REDIS und MySQL Datenbanken

Redis ist eine In-Memory Key-Value Datenbank. Im Gegensatz zu relationalen Datenbankmanagementsystemen (RDBMS) wie MySQL oder PostgreSQL werden in Redis keine festen Tabellenstrukturen hinterlegt. Redis gehört damit zur Kategorie der NoSQL Datenbanken (Not Only SQL). Key-Value Stores sind kein Ersatz für eine relationale Datenbank, bieten aber für bestimmte Bereiche große Vorteile. Durch das Fehlen von komplexen Strukturen innerhalb der Datenbank kann Redis Anfragen weitaus schneller als andere Datenbanksysteme bearbeiten. Da Redis im Random-Access Memory (RAM) ausgeführt wird, werden die Daten grundsätzlich nicht persistent gespeichert. ACID (Atomicity, Consistency, Durability and Isolation) Konformität wird mit Redis ebenfalls nicht

[9] Josheph, 2021.
[10] Grinberg, 2018.

gewährleistet. Für den Einsatzzweck als Cache in einer Cloud Umgebung ist Redis allerdings sehr gut geeignet.[11]

Innerhalb des Redis Key-Value Stores werden alle relevanten Daten gespeichert, die ein Nutzer während seiner Benutzung der Software produziert. Dort werden ebenfalls die Zwischenergebnisse abgespeichert, die die KI während der Analyse erstellt. MySQL ist ein um 1995 erschienenes Open-Source RDBMS. MySQL ist eines der weitverbreitetsten und schnellsten Datenbanksysteme in seiner Kategorie.[12]

In relationalen Datenbanken werden Daten strukturiert in Tabellenform abgespeichert. Einzelne Tabellen können Verlinkungen und Referenzen auf andere Tabellen haben, damit die Zusammengehörigkeit der Daten beschrieben werden kann, ohne Daten redundant speichern zu müssen. In MySQL, wie auch anderen RDBMS, werden Tabellenstrukturen und Daten persistent abgespeichert. In-Memory Datenbanken wie Redis können Daten über Umwege auch persistent speichern, jedoch müssen dafür größere Anpassung an der Konfiguration von Redis vorgenommen werden.

Das RDBMS MySQL wird unter Anderem für die Speicherung der Logs, die der Flask Server während der Verarbeitung von Requests oder Nachrichten an die KI produziert, verwendet. Ein weiterer Einsatzzweck der MySQL Datenbank ist die Speicherung der im System registrieren KI-Services. Ein Dienst kann über die Flask API im System registriert oder deregistriert werden. Das Frontend kann im Anschluss eine Auflistung der verfügbaren Services beim Backend anfragen.

2.5.3 Angular Frontend

Eine grundlegende Website wird klassisch mit Hypertext Markup Language (HTML) und JavaScript erstellt. Um eine moderne Website zu entwickeln, die ihren Inhalt nicht beim ersten Aufrufen lädt, sondern erst dann, wenn er benötigt wird, müssen Konzepte wie Asynchronous JavaScript and XML (AJAX) verwendet werden. Angular ist ein von Google entwickeltes und gepflegtes Open Source Framework, welches das Entwickeln von komplexen webbasierten Anwendungen vereinfachen soll. Angular bietet im Gegensatz zu anderen Webframeworks wie React und Vue.js eine vollumfängliche Bibliothek, mit der nahezu alle Aspekte in der Web Entwicklung abgedeckt werden können.[13]

In Angular wird die Programmiersprache TypeScript verwendet. Diese ist eine Erweiterung der Programmiersprache JavaScript und implementiert Konzepte wie feste Typisierung von Variablen. Weitere Konzepte wie Dependency Injection oder die Trennung von Business Logic (BL) und User Interface (UI) ermöglichen eine schnelle Entwicklung von komplexen Systemen.

Das Frontend wird für die Ein- und Ausgabe der Daten verwendet. Der Nutzer kann auf der Webseite seine Suchanfrage in ein Textfeld schreiben und anschließend auf den Server hochladen. Im nächsten Schritt wird die Möglichkeit bereitgestellt, die eingegeben Daten automatisiert zu bearbeiten und zu manipulieren. Im gleichen Zug wird die Eingabe des Nutzers in ein für die KI verständliches Format konvertiert. Im letzten Schritt kann der

[11] Paksula, 2010.
[12] DuBois, 2008.
[13] Moiseev u. a., 2018.

Nutzer die Anfrage an das Backend schicken, dass mit der Analyse der Eingabe begonnen werden soll. Das Frontend fängt daraufhin an beim Backend in regelmäßigen Abständen nach Antworten der KI zu fragen. Wenn Antworten vorhanden sind, können diese in einer Liste visualisiert werden.

2.5.4 Logging durch Grafana

Grafana ist ein von Torkel Ödegaard in 2014 entwickeltes Open-Source Datenvisualisierungsprogramm. Grafana kann zeitbasierte Daten in verschiedenen Arten von Grafen und Diagrammen anzeigen.[14]

Eines der möglichen Panels für ein Dashbaord ist das Log-Panel. Dort werden die Log Nachrichten aus einer Datenbank angezeigt und mit einer Farbe, abhängig vom Schweregrad markiert. Als Datenquelle können unter Anderem zeitbasierte Datenbanken wie InfluxDB und Prometheus oder RDBMS wie MySQL verwendet werden.

Im implementierten Prototypen wurde eine MySQL verwendet, in der die zu loggende Nachricht, der Schweregrad, ein Zeitstempel und die User Identifikation (ID) gespeichert werden. Diese Daten werden verwendet, um die Logs im Log-Panel von Grafana chronologisch anzeigen zu lassen.

2.5.5 Deployment über Docker

Docker ist eine Software zur Virtualisierung von Containern. Ein Container beschreibt eine in sich geschlossene Umgebung, in der ein Programm ausgeführt werden kann. Alle benötigten Dateien, Parameter und Umgebungsvariablen werden beim Starten des Containers mitgegeben. Damit kann sichergestellt werden, dass ein Programm, welches innerhalb eines Docker Containers ausgeführt wird, sich in jeder Umgebung gleich verhält. Hierdurch wird die Unabhängigkeit vom Host-Betriebsystem gewährleistet. Im Gegensatz zu einer Virtuelle Maschine (VM) muss für die Ausführung eines Docker Containers kein komplettes Betriebssystem virtualisiert werden. Das Hochfahren einzelner Container ist deutlich schneller und ressourcenschonender als die Implementierung einzelner VMs.[15]

Des Weiteren können über das Docker Compose Plugin mehrere Container gleichzeitig hochgefahren werden, sodass mit einer einzigen Kommandozeileneingabe eine komplette Softwarearchitektur hochgefahren werden kann.

Docker wird für das Deployment der einzelnen Komponenten des Prototyps verwendet. Für Redis, MySQL, RabbitMQ, Grafana und Elasticsearch können die benötigten Images, die eine Bauanleitung darstellen, aus dem Docker Hub heruntergeladen und genutzt werden. In einem Docker Image sind auch alle für die Ausführung des Programms benötigten Dateien gepackt. Docker Hub ist eine Plattform zur Verteilung von offiziellen Docker Images, von der automatisch alle Images runtergeladen werden, die lokal nicht vorhanden sind.

Für das Angular Frontend und das Flask Backend müssen die Images erst manuell gebaut werden, bevor sie als Container gestartet werden können. Dafür bietet die Docker

[14] Chakraborty u. a., 2021.
[15] Anderson, 2015.

sogenannte Dockerfiles an, in der die benötigten Konfigurationen hinterlegt werden können.

3 Methodik

text

3.1 Design Science Research

text[16]

3.2 Evaluationsmethode

text

[16] Frauchiger, 2017.

4 Konzeption und prototypische Umsetzung

In diesem Kapitel wird die prototypische Implementierung der Schnittstelle für die Anbindung von austauschbaren Datenquellen an KI-Algorithmen beschrieben.

4.1 Anforderungen

text

4.2 Konzeption

4.2.1 Softwarearchitektur

Ein grundlegender Dienst, der Daten mit einer KI verbindet, kann mithilfe eines einzigen Python-Scripts erstellt werden. Die Herausforderung an einer praxistauglichen Anwendung, die gleichzeitig von mehreren Usern genutzt werden kann, liegt im Architekturdesign der Software. Eine praxistaugliche Anwendung muss neben den funktionalen Anforderungen auch noch weitere nicht funktionale Anforderungen erfüllen. Die drei wichtigsten nicht funktionalen Anforderungen sind Performance, Skalierbarkeit und Verfügbarkeit. Alle drei Anforderungen können mit einem lokal ausgeführten Skript nicht erfüllt werden.

Damit Nutzer mit der Software interagieren können, wird ein Frontend benötigt. Ein zentral gehostetes, webbasiertes Frontend kann von einem Nutzer über eine einfache URL im Webbrowser aufgerufen werden. Für die anzuzeigenden Daten im Frontend wird eine Verbindung zum Backend benötigt. Diese wird über eine HTTP Verbindung zur mit Flask gehosteten REST API bereitgestellt.

Um die Anforderung der Skalierbarkeit erfüllen zu können, ist die REST-API komplett zustandslos implementiert worden. Eine API ohne Zustände speichert keine Zwischenstände zu den Anfragen einzelner Nutzer. Bei jeder Anfrage an die API müssen alle Informationen im Request bereitgestellt werden, die die API zum bearbeiten der Anfrage benötigt. Dies bietet die Möglichkeit bei steigender Nutzerzahl mehrere parallel betriebene Instanzen der API hochzufahren. Dadurch ist eine horizontale Skalierung gewährleistet. Horizontal skalierbare Instanzen innerhalb der Software Architektur sind in Abbildung 1 mit zwei hintereinander gestapelten Rechtecken visualisiert.

Da die Kommunikation zwischen dem Frontend, der API und den KI-Services asynchron läuft, muss das Flask Backend trotz seiner Zustandslosigkeit Transformationsanleitungen und Ergebnisse der KI-Services zwischenspeichern, bis sie im Frontend benötigt werden. Um die Performanceanforderungen erfüllen zu können, können nicht alle Zwischenstände in einer MySQL Datenbank gespeichert werden. Die Lese- und Schreibgeschwindigkeit kann bei steigender Nutzerzahl problematisch werden. Um dem Entgegenzuwirken wird ein Redis Key-Value Store als Cache betrieben. Die zwischengespeicherten Daten werden nach dem ersten Aufruf wieder gelöscht, weswegen eine persistente Speicherung nicht notwendig ist. In-Memory Datenbanken speichern und führen ihre Queries direkt im RAM aus, wodurch Anfragen im Vergleich zu einer MySQL Datenbank deutlich schneller ausgeführt werden.

Im Flask Backend werden alle Routen und die meisten Funktionen abgekapselt in einem Funktion Wrapper ausgeführt. Dieser fungiert als eine Art Sandbox, in der auftretende

Fehler nicht zum Programmabsturz führen, sondern behandelt und geloggt werden können. Alle Logs werden persistent in einer MySQL Datenbank gespeichert. Mit dem Dienst Grafana können diese Logs angezeigt werden.

Die Laufzeit von KI-Services kann sehr stark vom verwendeten KI-Modell, der zu durchsuchenden Datenmenge, wie auch der vom Nutzer gesendeten Eingabe abhängen. Bei einer synchronen Kommunikation zwischen dem Flask Backend und dem Service können sehr lange Wartezeiten entstehen. Wenn der KI-Service ebenfalls eine REST-Schnittstelle implementieren würde, könnten es bei einem HTTP Request zum Timeout der Anfrage führen. Aufgrund der schwanken Laufzeit muss eine asynchrone Kommunikationsstruktur, wie RabbitMQ mit dem AMQP implementiert werden.

Die einzelnen Services können mit einem Eintrag in der MySQL Datenbank registriert werden. Für die Registrierung muss lediglich der Name und der im Frontend anzuzeigende Name des Services hinterlegt werden. Die Registrierung eines Dienstes kann durch den Aufruf einer Route in der API durchgeführt werden.

Der im Prototypen implementierte KI-Service nutzt das BERT Modell von Google zum konvertieren der Nutzereingaben in semantische Vektoren. Es wird ebenfalls eine Elasticsearch Datenbank betrieben, in der alle zu Durchsuchenden Einträge gespeichert sind. Im Gegensatz zu einer MySQL Datenbank, kann in einer Elasticsearch Datenbank zu jedem Eintrag ein semantischer Vektor gespeichert werden. Der KI-Service kann mithilfe der Kosinusähnlichkeitssuche den semantischen Vektor der Eingabe mit den Vektoren der Datenbank vergleichen und so die semantisch ähnlichstes Texte herausfiltern. Die gefunden Einträge werden über RabbitMQ im Anschluss wieder an das Flask Backend geschickt, damit sie dort vom Frontend ausgelesen werden können.

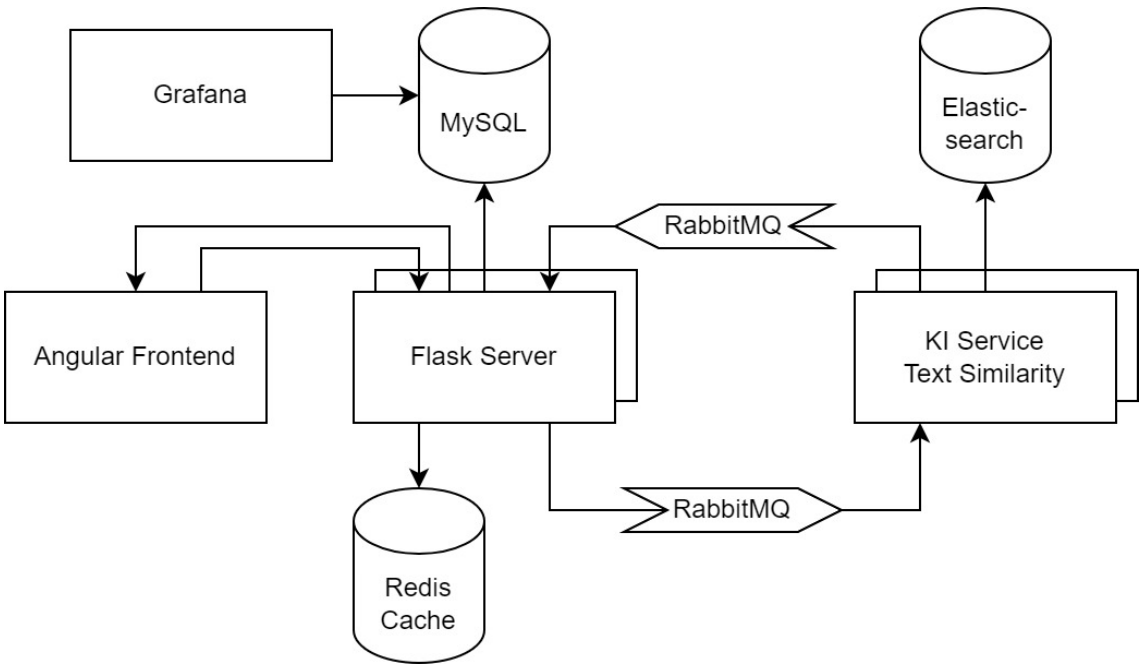

Abbildung 4: Softwarearchitekturdiagramm

4.2.2 Programmablauf

text

4.2.3 KI-Services

text

4.3 Prototypische Umsetzung

text

4.3.1 Implementierung der REST-API

Python bietet mit dem Package Flask die Möglichkeit einen simplen und gut skalierbaren Webserver aufzusetzen. Für das Starten einer Flask Instanz muss das Package Flask in die Python Umgebung importiert werden. Anschließend kann ein Flask-Objekt erzeugt und die Flask Instanz mit den gewünschten Parametern gestartet werden.

```
from flask import Flask
app = Flask(__name__)
app.run(host="0.0.0.0", port=80, use_reloader=False)
```

Damit die API auch automatisiert aus einem Docker Container heraus gestartet werden kann, muss die Ausführung des Flask Services in die Main Methode von Python ausgelagert werden. Flask blockiert den Thread auf dem es ausgeführt wird, was eine asynchrone Kommunikation über RabbitMQ nicht möglich macht. Der Receiver benötigt seinen eigenen Thread, weswegen eine Multithreading-Architektur implementiert werden muss. Zu diesem Zweck wird das threading Package genutzt. Über den Parameter `daemon` kann bei der Erzeugung eines Threads festgelegt werden, dass der Thread im Hintergrund läuft und den Hauptthread nicht blockiert.

```
def start_server():
    app.run(host="0.0.0.0", port=80, use_reloader=False)

if __name__ == '__main__':
    thread_server = threading.Thread(target=start_server, daemon=True).
       start()
```

Eine in der API adressierbare Route kann in Flask über Function-Annotations definiert werden. Die von Flask implementierten Anntotations haben die Form `instanz.route('path', methods=["METHOD"])`. Der Name der Instanz wird am Anfang des Projekts als `app` definiert. Der `path` beschreibt die Route, die vom Frontend aufgerufen werden muss, damit die nachfolgende Funktion ausgeführt wird. Im Array `methods` besteht die Möglichkeit, ein oder mehrere Request Typen zu definieren, die die Funktion akzeptieren soll.

Eine beispielhafte Nutzung der Annotations, um eine Route in der API zu definieren, ist nachfolgend aufgeführt.

```
@app.route('/', methods=["GET"])
def index():
    [..]
    return r.respond({"token": token}, cookie=f"Authorization={token}")
```

Der Inhalt der Methode und deren genaue Funktionsweise wird in den folgenden Kapiteln näher erläutert.

Die Funktion `respond` ist im Skript `api/response_generator.py` definiert. Sie dient als Function Wrapper, der bei jeder ausgehenden Response die Response-Header, eventuelle Cookies und den Reponse Typen setzt. Der Output der Response wird mithilfe des `json` Packages in JSON Syntax konvertiert.

```
import json
from flask import Response
def respond(r, status=200, json_dump=True, cookie=""):
    [...]
    return Response(json.dumps(r), status=status, mimetype='application
        /json', headers=headers)
```

In Tabelle 1 sind alle in der API verfügbaren Routen aufgelistet. Auf die genaue Funktionalität der einzelnen Funktionen wird in den folgenden Kapitel eingegangen.

Route	Typ	Funktion
'/'	GET	Erstellung eines JSON Web Tokens
'/upload/file'	POST	Hochladen einer Textdatei für den Input der KI
'/upload/text'	POST	Texteingabe für den Input des KI-Services
'/transform'	POST	Festlegen der Tranformationseigenschaften
'/send'	POST	Transformieren und Senden des Inputs an einen KI-Service
'/poll'	GET	Abfrage der vom KI-Service gelieferten Ergebnisse
'/service'	GET	Auflistung aller Services
'/service'	POST	Registrieren eines neuen Services
'/service'	DELETE	Löschen eines Services

Tabelle 2: Implementierte Routen der REST-API

4.3.2 Nutzeridentifizierung mit JWT

Innerhalb des Backendes ist es notwendig, einzelne Nutzer voneinander zu unterscheiden. Für jeden Nutzer speichert das Backend den hochgeladenen Text, die Transformationsanleitung und die Antworten des angefragten KI-Services im Redis Cache. Um Nutzer voneinander unterscheiden zu können, gibt es zwei grundlegende Möglichkeiten.

1. Identifizierung durch den Nutzer der Software. Beispielsweise mittels Registrierung durch E-Mail Adresse und Passwort.
2. Identifizierung durch das Backendend der Software. Generierung und Zuweisung einer zufälligen aber eindeutigen User-ID.

Die Erhebung von personenbezogenen Daten setzt die Einhaltung der Datenschutz-Grundverordnung (DSGVO) voraus. Dies bedeutet einen erheblichen Mehraufwand für eine Anwendung, die sonst keinen weiteren Nutzen aus den Daten zieht.

Das Backend nutzt einen Universally Unique Identifier (UUID) der sich durch das Python Package `uuid` generieren lässt. Eine UUID ist eine 32 Zeichen lange Zahl im Hexadezimalformat. Die importierte Funktion `uuid4()` erzeugt eine zufällige, ohne von Parametern beeinflusste UUID. Der Nutzer muss diese UUID mitgeteilt bekommen und für alle seine Anfragen, aufgrund der zustandslosen Implementierung der API, im Authorization Hea-

der mitschicken. Damit die UUID nicht ausgelesen oder manipuliert werden kann, wird sie nicht als einfacher Text in der Response an den Nutzer geschickt, sondern vorher in ein JSON Token geschrieben und verschlüsselt.

Ein JSON Web Token (JWT) ist ein kompaktes, URL-sicheres Mittel zur Darstellung von Forderungen, die zwischen zwei Parteien übertragen werden sollen. Die Angaben in einem JWT werden als JSON-Objekt kodiert. Der Inhalt des JST kann digital signiert oder die Integrität mit einem Message Authentication Code (MAC) geschützt und/oder verschlüsselt werden.[17]

Im nachfolgenden Codeausschnitt ist die Generierung der UUID und die Verschlüsselung des JWT dargestellt.

```
def uuid_gen():
    return uuid.uuid4()

def encode_token(param):
    return jwt.encode(param, JWT_PASSWORD, algorithm="HS256")

token = encode_token({'uid': str(uuid_gen())})
```

Für jede Route, ausgenommen die `/service` Routen zum Management der Services, wird der JWT für die Ausführung benötigt. Die Überprüfung und Entschlüsselung des Tokens ist für jede Route gleich, daher ist es sinnvoll diese Funktionalität zu zentralisieren. Damit wird die Fehleranfälligkeit reduziert und und die Wartbarkeit erhöht, sollte sich zum Beispiel der Algorithmus oder das Passwort für die Verschlüsselung ändern. Wie auch Flask Annotations zum definieren einer Route verwendet, ist es möglich eigene Annotations zu entwerfen. Für diese Funktion ist das Python Package `functools` mit der Funktion `wraps` zuständig. Wraps ermöglicht es, Funktionen ineinander zu verschachteln.

Im Prototypen wird die Funktion `token_required(f)` definiert. Diese Funktion dient als eine Umgebung, in der eine weitere Funktion ausgeführt werden. Im Gegensatz zur normalen Ausführung einer Funktion, werden in der `token_required(f)` Funktion vor der Ausführung der eigentlichen Funktion mehrere Rahmenbedingungen geprüft. Der vom Nutzer gesendete Token muss nach der erfolgreicher Entschlüsselung syntaktisch korrektes JSON enthalten. Sollte dies nicht der Fall sein, wird die die eigentliche Funktion, die zur API Route gehört, gar nicht erst ausgeführt. Der Nutzer bekommt direkt eine Response mit dem HTTP Error-Code 401: Unauthorized gesendet.[18]

Wenn die Entschlüsselung des Tokens erfolgreich war, wird die innere Funktion ausgeführt. Als Parameter der inneren Funktion wird die im JWT enthaltene UUID übergeben. Durch diesen Aufbau ist der Code für die Verifizierung des Tokens und die Logik der Funktion unter der angesprochenen Route vollständig getrennt.

```
@routes.route('/upload/text', methods=['POST'])
@token_required
def upload_text(uid):
    [...]
```

[17] Jones u. a., 2015.
[18] R. Fielding u. a., 1999.

4.3.3 Caching mit Redis Datenbank

Redis ist ein Key-Value Store der vollständig im RAM ausgeführt wird. Innerhalb von Redis sind mehrere Datenbanken definiert, die in ihrer Standartkonfiguration über einen Index i, mit $0 \leq i < 16$ aufgerufen werden. Im Backend werden die ersten drei Datenbanken verwendet.

1. Datenbank 0: Cache der hochgeladenen Textdateien für den Input der KI
2. Datenbank 1: Cache der Transformationsanleitung
3. Datenbank 2: Cache der vom KI-Service produzierten Ergebnisse

4.3.4 Kommunikation zwischen Backend und Services mit RabbitMQ

Für den Informationsaustausch zwischen dem Backend und den verschiedenen KI-Services ist eine asynchrone Kommunikation implementiert. Je nach Komplexität des Services, kann die Verarbeitung einer vom Nutzer gestellten Anfrage mehrere Sekunden bis Minuten dauern. Eine synchrone Kommunikation, in der der Client auf unbestimmte Zeit auf eine Antwort wartet ist nicht möglich. Wenn nach einer vom Browser definierten Zeit keine Antwort auf den Request kommt, wird der Request mit einem Timeout abgebrochen. Sollte die KI nach der maximal verfügbaren Zeit ihr Ergebnis liefern, wird dieses Verworfen und ner Nutzer muss eine neue Anfrage stellen. Damit Anfragen nicht verloren gehen und die Antworten dem Server mitgeteilt werden können, wenn sie bereit sind, wird der Message Broker RabbitMQ implementiert.

RabbitMQ dient als Middleware, die Anfragen vom Server annimmt und diese im in einer Queue zwischenspeichert, um sie anschließend an die Services zu verteilen. Damit die Nachrichten in eine Queue geschrieben werden können, muss im ersten Schritt eine Verbindung zu RabbitMQ aufgebaut werden. Mithilfe des Packages `pika` kann über den Host, unter dem RabbitMQ erreichbar ist, den Port 5672 und den Login-Credentials eine Transmission Control Protocol (TCP) Verbindung aufgebaut werden.

Innerhalb einer Connection, können über einen Channel können sowohl der Server, als auch die Services eine Verbindung mit dem RabbitMQ Dient aufbauen. Ein Channel beschreibt die logische Verbindung zwischen dem Server/Service und dem Broker. Für jede TCP Verbindung mit RabbitMQ können mehrere Channels erstellt werden.[19]

In einem Channel wird letztendlich die Queue definiert, in der die Nachrichten zwischengespeichert werden. Im folgenden Codeausschnitt ist gezeigt, wie eine Connection, ein Channel und eine Queue erstellt werden.

```
def produce(uid, service, query, params):
    connection = pika.BlockingConnection(pika.ConnectionParameters(
        rabbit_host, 5672, '/', credentials))
    channel = connection.channel()
    channel.queue_declare(queue=service)
```

Die Nachrichten, die der Server an die Services schickt, werden mittels der im Channel definierten Methode `basic_publish` in die für den Service entsprechende Queue geschrieben werden. Im vorherigen Codeausschnitt wird der Parameter `service` in der `produce`

[19] Dossot, 2014.

Methode übergeben. Dieser ist abhängig vom Service, an den die Anfrage gerichtet ist. Im Prototypen wurde der Service `text-similarity` implementiert. Dieser kann im Frontend als Ziel ausgewählt werden. Wenn der Nutzer im Frontend die Anfrage an den `text-similarity` Service stellt, wird die Methode `produce` mit dem String „`text-similarity`" aufgerufen.

Sollte es die Queue mit dem aufgerufenen Service noch nicht geben, wird diese von RabbitMQ beim ersten Aufrufen erstellt. Ist sie bereits vorhanden, wird lediglich ein Eintrag in die entsprechende Queue geschrieben.

Die Nachricht, welche aus der Anfrage des Nutzers, den eingegebenen Parametern und weiteren im Backend automatisch generierten Parametern, wie dem UUID, besteht, wird anschließend mittels AMQP an den Rabbit Broker gesendet. Dort wird die Nachricht, wie zuvor beschrieben, in die entsprechende Queue eingetragen.

Auf Seiten des Services wird eine Methode implementiert, die die Nachrichten in einer Queue auslesen kann. Zu beginn muss eine Queue definiert werden, aus der Nachrichten ausgelesen werden. Dies funktioniert analog zum erstellen einer Queue für das Schreiben von Nachrichten über `queue_declare` gefolgt von dem Namen der Queue. Das Registrieren als Consumer für eine Queue wird mittels der Methode `basic_consume()` durchgeführt. Innerhalb der Parameter ist eine Callback Methode definiert, die ausgelöst wird, wenn der Service eine Nachricht erfolgreich aus der Queue ausgelesen hat. Die `callback` Methode ist der Startpunkt des Services, von dem aus die eingehende Nachricht mithilfe der KI analysiert und verarbeitet wird.

Um den Consumer zu starten und damit auf neue Nachrichten in der Queue zu warten, wird die Methode `start_consuming()` ausgeführt. Im nachfolgenden Codesegment ist der eben beschriebene Prozess im KI-Service aufgeführt.

```
channel.queue_declare(queue='text-similarity')
channel.basic_consume(queue='text-similarity',
                      auto_ack=True,
                      on_message_callback=callback)
channel.start_consuming()
```

Nachdem die KI die eingehende Anfrage verarbeitet hat, schreibt der Service die Response in eine Queue. Es kann jedoch nicht die gleiche Queue für Anfragen und Antworten verwendet werden, da es sonst dazu führen könnte, dass vom Service produzierte Antworten wieder als Anfrage interpretiert werden und es so zur fehlerhaften Ausführung des KI-Algorithmus kommt. Ein weiteres Problem wäre, dass die Antworten dann aus der Queue herausgenommen wurden und der Server keine Antwort mehr zur gestellten Anfrage erhält. Um dem entgegenzuwirken wurde eine zweite „Response-Queue“ implementiert, in die ausschließlich die Antworten des Services geschrieben werden. Da die Queues über ihren Namen definiert werden, hat jede Response Queue einen vom Service abhängigen, automatisch generierten Namen. Der Name hat die Form `response-{service}`. Im Falle des `text-similarity` Services, hat die Antwort Queue den Namen `response-text-similarity`.

```
channel.queue_declare(queue='response-text-similarity')
```

```
channel.basic_publish([...], body=f'{{"uid": "{uid}", "service": "{
    service}", "message": {json.dumps(message)}}}'.encode('utf-8'))
```

Der Server implementiert, ähnlich wie der Service auch, eine Möglichkeit die Nachrichten in der Response Queue auszulesen. Innerhalb der Nachricht, die an den Service geht und vom Service wieder zurück kommt, wird die UUID des Nutzers mitgeliefert. Damit ist eine anschließende Zuordnung von Anfrage und Antwort möglich. Das Backend speichert die Antwort des KI-Services im Redis Cache und kann sie dem Frontend bei Bedarf zur Verfügung stellen.

Der Ablauf der Kommunikation zwischen dem Server und den Services mit RabbitMQ ist in Abbildung 4 veranschaulicht.

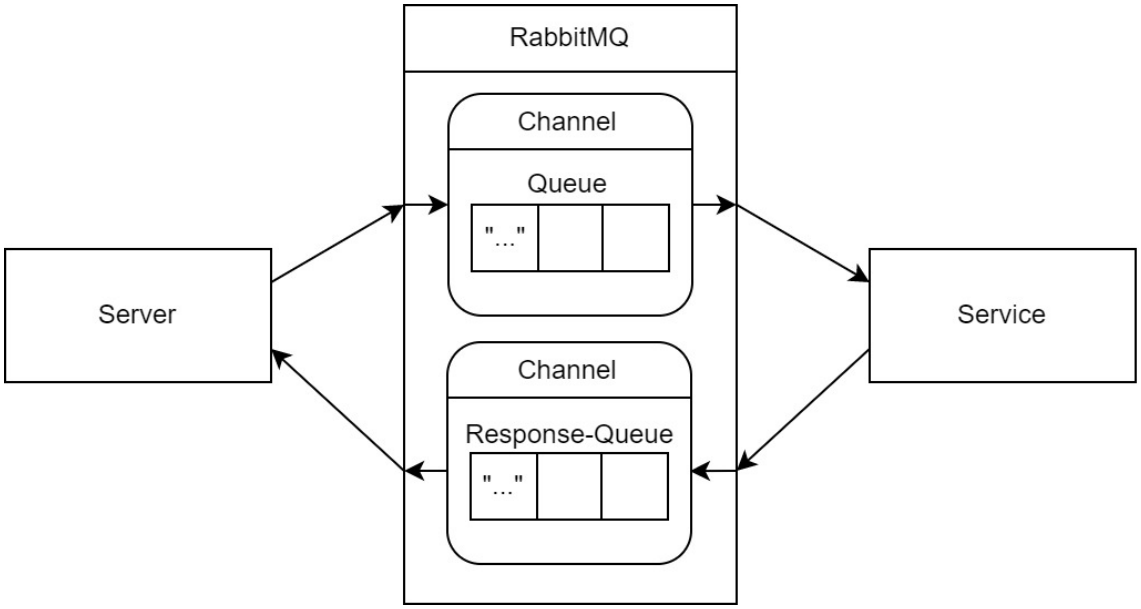

Abbildung 5: Kommunikation mit RabbitMQ

4.3.5 Textähnlichkeitssuche mit Elasticsearch

Für den Prototypen wurde ein KI-Service zur Textähnlichkeitssuche entwickelt. Dieser Service ermöglicht eine semantische Suche in einer Datenbank. Nicht-KI gestützte Textsuchalgorithmen können in einer Datenbank lediglich nach übereinstimmenden Wörtern oder Wortteilen suchen. Wenn ein Nutzer alle Einträge zu einem bestimmten Thema aus der Datenbank herausfiltern möchte, muss die passenden Schlüsselwörter kennen, die in den Datenbankeinträgen vorhanden sind. Such man in einer Datenbank beispielsweise nach „JavaScript“ und es gibt nur Artikel, die die Abkürzung „JS“ beinhalten, liefert die Suche keine Ergebnisse. Eine gewisse Textähnlichkeit kann erreicht werden, wenn zum Beispiel Leerzeichen oder Groß/Kleinschreibung ignoriert werden.

Eine Textänlichkeitssuche, die mittels KI durchgeführt wird, kann zu einer Anfrage wie „JavaScript“ deutlich mehr Ergebnisse liefern. Sie ist nicht durch die in der Suche angegebenen Zeichen beschränkt, sondern versucht den Inhalt der Anfrage zu verstehen. Anschließend kann in der Datenbank nach inhaltlich ähnlichen Artikeln gesucht werden, die zur Anfrage passen. Der „verstandene“ Satz, den man für die Suche in der Datenbank nutzt, ist abhängig vom verwendeten KI Modell. Für die Textähnlichkeitssuche im

Prototypen wird das BERT Modell von Google verwendet, um die Anfrage in einen 512-dimensionalen Vektor zu konvertieren. Jede Dimension des Vektors wirkt sich auf die Interpretation des Vektor aus. Der Vektor wird daher auch „dense vector“, oder auch dichter Vektor genannt. Damit die Suche über den Vektor funktioniert, muss es einen zweiten Vektor geben, mit dem der, durch die Suchanfrage erzeugte, Vektor verglichen werden kann. Jeder Eintrag in der Datenbank muss daher einmalig mit dem gleichen KI-Modell analysiert werden. Der jeweils erzeugte Vektor wird ebenfalls in der Datenbank abgespeichert, um ihn für kommende Suchanfragen nutzen zu können. Dieser Prozess ist in Abbildung 5 unter „Daten indizieren“ aufgeführt.

Der Grundsatz des KI-Modells ist es, dass zwei Sätze, die inhaltlich ähnlich sind, auch zwei ähnliche Vektoren besitzen. Vektoren sind dann ähnlich, wenn der Abstand der beiden gering und die Richtung ähnlich ist.[20] Diese Abstandsberechnung kann in der Elasticsearch Datenbank seit Version 7.3, welche am 31.07.2019 veröffentlicht wurde, automatisch durchgeführt werden.

Nachdem alle Einträge in der Elasticsearch Datenbank indiziert wurden, beginnt die Programmablaufschleife. Der erste Schritt der Schleife ist das warten auf eingehende Nachrichten. Diese Nachrichten werden über RabbitMQ aus der, für den Service definierten, Queue ausgelesen. Falls keine Anfrage an den Service eingetroffen ist, wartet der Dienst auf unbestimmte Zeit. Nach dem erfolgreichen Auslesen einer Nachricht, fängt der KI-Service an, die Anfrage zu bearbeiten. Im ersten Schritt wird aus dem Satz, den der Nutzer an den Service geschickt hat, ein semantischer Vektor generiert. Dazu wird das gleiche KI-Modell wie bei der Indizierung der Einträge in der Datenbank genutzt. Daraufhin nutzt der Service den generierten Vektor und die `cosineSimilarity()` Funktion der Elasticsearch, um die semantisch ähnlichsten Texte aus der Datenbank herauszufiltern. Die Anzahl der gelieferten Ergebnisse ist abhängig von den übergebenen Parametern. Der Nutzer kann im Frontend das Key-Value Paar `search_size` eingeben, welches in der Anfrage an den Service mitgeliefert wird. Sollte der Nutzer beispielsweise „`search_size : 3`" eingeben, so werden die drei ähnlichsten Artikel zur gestellten Anfrage zurückgegeben. Die Ergebnisse werden in die Response Queue des Services geschrieben, damit sie im Backend weiter verarbeitet werden können. Nach erfolgreichem Durchlaufen des Prozesses, geht der Service wieder zum Zustand „Auf Anfrage warten“ über. Veranschaulicht wird der Programmablauf des Services in Abbildung 5.

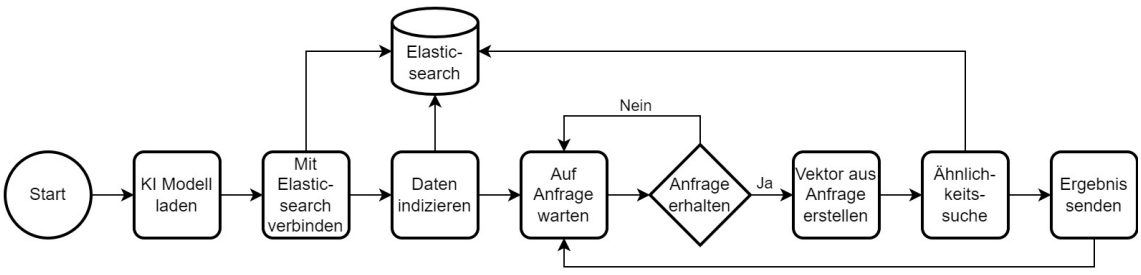

Abbildung 6: Ablaufdiagramm der Textähnlichkeitssuche

[20] Rahutomo u. a., 2012.

4.3.6 Implementierung des KI-Services

Damit der KI-Service eine Textähnlichkeitssuche durchführen kann, muss zunächst das BERT Modell importiert werden. Dieses kann der Service beim Starten automatisch aus dem offiziellen Tensorflow Hub herunterladen. Anschließen wird eine Tensorflow Session erstellt und gestartet.

```
embed = hub.Module("https://tfhub.dev/google/universal-sentence-encoder
    /2")
```

Im zweiten Schritt verbindet sich der Service mit der Elasticsearch Datenbank. Dazu wird die URL, unter der die Elasticsearch erreichbar ist aus den Environment Variablen geladen. Die Environment Variablen können entweder beim manuellen Starten des Services mitgegeben werden, oder falls der Service mit Docker gestartet wird, in der Docker-Compose File definiert werden. Mithilfe der URL initialisiert der Service eine Elasticsearch Instanz und speichert sie in der Variable `client` ab.

```
elasticsearch_url = os.environ.get('ELASTIC_URL')
client = Elasticsearch(elasticsearch_url)
```

Nachdem sich der Service erfolgreich mit der Elasticsearch verbunden hat, beginnt das Indizieren der Daten in der Datenbank. Dazu wird die Methode `index_data()` aufgerufen. Der Datensatz, der für den Prototypen verwendet wurde, ist ein Auszug aus den gestellten Fragen im Forum Stackoverflow. In dem Datensatz sind 18.600 Fragen vorhanden. Jede Frage besteht aus einem Titel und einem Body. In einem Fragen-Body ist eine genauere Erläuterung der Frage, mit eventuellen Codeausschnitten oder Verlinkungen. Für die Indizierung ist lediglich der Titel der Frage genutzt worden. Das bedeutet im Umkehrschluss, dass der Titel der Frage repräsentativ für die gesamte Frage stehen muss. Nach Möglichkeit soll der Titel den gesamten Inhalt der Frage kurz und bündig zusammenfassen. Aus diesem Grund eignet Stackoverflow sich besonders gut für den Datensatz, da in den Guidelines „Write a title that summarizes the specific problem“ explizit erwähnt wird.[21]

Nach erfolgreicher Indizierung aller Fragestellungen, geht der Service in die Programmablaufschleife über. Die Schleife beginnt mit der Initialisierung der RabbitMQ Verbindung. Dort Verbindet sich der der Textähnlichkeitsservice mit der Queue `text-similarity`. Bei einer eingehenden Anfrage ruft die `callback` Funktion die `respond` Funktion auf, die den KI Algorithmus aufruft und sich um die Verarbeitung der Antwort kümmert. Die Funktion `handle_query()` benötigt als Parameter die Query die vom Nutzer gestellt wurde und die Anzahl der Ergebnisse, die zurückgegeben werden sollen. Beide Parameter werden aus aus der Anfrage, die über RabbitMQ an den Service gesendet wurde, ausgelesen. Sollte es vorkommen, dass der Nutzer keine `search_size` angegeben hat, wird der Standartwert 1 gesetzt.

```
search_size = 1
    for p in params:
        if "search_size" in p.values():
            search_size = p["value"]
```

[21] Stackoverflow, 2022.

```
    message = handle_query(query, search_size)
```

Innerhalb der `handle_query()` Methode wird zunächst der Satz, der vom Nutzer gesendet wurde, durch die Methode `embed_text()` in einen Vektor konvertiert. Dieser wird anschließend in das Feld „`params`" in der Such-Query für die Elasticsearch geschrieben. In dem Feld „`source`" wird die eigentliche Ähnlichkeitsanalyse definiert. Die Elasticsearch implementiert die Funktion `cosineSimilarity()`, die zwei Vektoren als Parameter annimmt. Der erste Vektor ist der aus der Query zuvor generierte Vektor. Der zweite Vektor ist der aus dem Titel der Frage generierte Vektor.

Mit der `client.search()` stellt der Service die Anfrage an die Elasticsearch mit der zuvor definierten Query. In der Anfrage wird über das Feld „`size`" die Anzahl der gewünschten Ergebnisse mitgeliefert. Wenn die `search_size` beispielsweise 3 beträgt, werden die drei Fragen mit den ähnlichsten Vektoren in die Variable `response` geschrieben. Im nachfolgen Codesegment sind die zuvor beschriebene Erstellung des Query Vektors, der Such-Query für die Elasticsearch und die Anfrage an die Elasticsearch dargestellt.

```
query_vector = embed_text([query])[0]
script_query = {
    [...]
        "source": "cosineSimilarity(params.query_vector, doc['
            title_vector']) + 1.0",
        "params": {"query_vector": query_vector}
}
response = client.search(
    [...]
        body={
            "size": search_size,
            "query": script_query,
            [...]
        }
)
```

Die Response der Elasticsearch wird nach erfolgreicher Durchführung wieder an die `respond` Funktion übergeben. Dort wird die Nachricht vorbereitet, die als Response auf die Anfrage vom Backend zurückgesendet wird. Die Antwort wird an RabbitMQ in die Queue `response-text-similarity` gesendet.

Der Schleifenduchlauf des Services ist damit abgeschlossen. Der Service geht nun wieder, wie in Abbildung 5 zu sehen, zum Zustand „Auf Anfrage warten“ über.

4.3.7 Management der Services

Die entworfene Softwarearchitektur unterstützt eine beliebige Anzahl an KI-Services. Alle Services, die über das System erreichbar sein sollen, müssen über die API registriert werden. Jeder Service implementiert eine Verbindung zur RabbitMQ Schnittstelle, damit die Anfragen und Antworten asynchron zwischen dem Backend und dem jeweiligen Service gesendet werden können. Eine der Kernanforderungen für eine Architektur, die austauschbare KI-Services unterstützt, ist die dynamische Registrierung und Deregistrierung. Eine Registrierung beschreibt das Hinzufügen eines neuen Services im System,

ohne den Quellcode anpassen zu müssen. Um dies zu ermöglichen, braucht es einen persistenten Speicher, in dem alle zur Verfügung stehenden Services aufgelistet sind. Zur persistenten Speicherung von kleineren Datenmengen, eignet sich eine MySQL Datenbank. Innerhalb der Datenbank werden alle aktiven Services in der Tabelle `services` abgespeichert. Zu jedem Service gibt es einen Namen, `servie`, und einen Anzeigenamen, `display_name`. Der Name wird für das interne Routing über RabbitMQ genutzt. Der Anzeigename wird im Frontend innerhalb des Dropdowns für die Auswahl des Services verwendet.

4.3.8 Automatisierte Transformation des Inputs

text

4.3.9 Fehlerbehandlung

Während des Laufzeit des Programms kann es dazu kommen, dass im vom Nutzer produzierte Fehler auftreten. Der Nutzer kann im Bereich der Transformation syntaktisch nicht korrekte Texte eingeben, die Backend verarbeitet werden. Da Nutzereingaben ohne weitere Behandlung ebenfalls ein Sicherheitsrisiko für die Infrastruktur darstellen können, wird im Backend ein System implementiert, um die Verarbeitung der Eingaben in einer isolierten Umgebung ausführen zu können. Ähnlich wie bei der Verifizierung des JWT, ist das System zur Fehlerbehandlung auch mit Function Wrappern und Annotations umgesetzt.

Im Backend wird ein Exception Handler definiert der eine Funktion mit zuvor übergebenen Parametern ausführt. Da eine fehlerhafte Ausführung auch dort zum Programmabsturz führen würde, wird die Funktion in einem try-except Block gekapselt. Alle innerhalb dieses Blocks auftretende Fehler werden abgefangen und in einem Exception Objekt gespeichert. Innerhalb des Exception Objekts ist die produzierte Fehlermeldung gespeichert. Über `str(e)` kann auf die Fehlermeldung zugegriffen werden. Innerhalb des Except Blocks wird die Fehlermeldung an den Logger weitergegeben, um diese in einer Datenbank persistent zu speichern. Nach erfolgreichem Log wird dem Frontend in einer HTTP Response der Statuscode 500 „Internal Server Error“ zurückgegeben.[22]

```
[...]
 @wraps(f)
        def decorator(*args, **kwargs):
            try:
                return f(*args, **kwargs)
            except Exception as e:
                logger.log("error", f"[Server, {msg}]: {str(e)}", "none
                    ")
                return r.respond({"success": False, "error": str(e)},
                    status=500)
            [...]
```

[22] R. Fielding u. a., 1999.

Jede Funktion, die in innerhalb des Exception Handlers ausgeführt werden soll, wird mit der Annotation `@exception_handler("...")` versehen. Innerhalb des Parameters, wird der Ort, an dem die Funktion ausgeführt wird, und damit der Fehler auftritt, als String übergeben. Diese Information wird genutzt, um innerhalb des Fehlerlogs den Ort des Fehlers aufzulisten.

4.3.10 Event Logging

Das Backend implementiert ein Event Logging System mit dem Zustände und Informationen des Systems in einer Datenbank gespeichert werden können. Mithilfe von Logs können Entwickler den Ablauf eines Programms besser nachvollziehen und auftretende Fehler schneller zu ihrer Quelle zurückverfolgen. Es ist ebenfalls möglich, Systeme aufzusetzen, die die auf Logs auslesen und beim Auftreten eines Errors die zuständigen Personen alarmieren. Eine Herausforderung bei einem Logging System ist es, dass das System dem Entwickler beim Loggen keinen erheblichen Mehraufwand produzieren soll. Im Backend wurde ein Logging System entwickelt, welches mit einer einzigen Funktion angesteuert werden kann. Die Log Funktions des Loggers wird in den verschiedenen Bereichen der Anwendung über `from logs.logger import log` importiert. Nach erfolgreichem Import, steht die `log` Funktion zur Verfügung.

Für einen Log müssen drei Parameter übergeben werden, der Log Level, eine Message und die UUID. Die möglichen Ausdrücke für die Log Level sind in Tabelle 2 aufgeführt. Die unterstützten Log Level leiten sich aus der Funktionalität von Grafana ab, welche diese Logs mit einer zugeordneten Farbe visualisiert.

Ausdruck	Log Level	Farbe
emerg	critical	lila
fatal	critical	lila
alert	critical	lila
crit	critical	lila
critical	critical	lila
err	error	rot
eror	error	rot
error	error	rot
warn	warning	gelb
warning	warning	gelb
info	info	grün
information	info	grün
notice	info	grün
dbug	debug	blau
debug	debug	blau
trace	trace	hellblau
*	unknown	grau

Tabelle 3: Log Level des Event Logging Systems

Innerhalb der Log Funktion wird eine Datenbank Query mit den drei Parametern und einem aktuellen Zeitstempel erstellt. Der Zeitstempel kann mithilfe des `datetime` Packages erstellt werden. Über `cursor.execute()` wird die erstellte Query in der MySQL Datenbank ausgeführt. Der Log wird als Eintrag in der Tabelle `logs` gespeichert.

```
def log(level, message, uid):
    [...]
    query = f"INSERT INTO logs (`level`, `message`, `timestamp`, `uid`)
        VALUES (%s, %s, %s, %s)"
    cursor.execute(query, (level, message, datetime.utcnow(), str(uid))
       )
    [...]
```

4.3.11 Website mit Angular

text

4.4 Deployment der Software mit Docker

text

5 Evaluation

text

5.1 Performanceanalyse

text

5.2 Skalierbarkeit

text

5.3 Ergebnisse des Code-Reviews

text

6 Fazit und Ausblick

6.1 Fazit

6.2 Implikation für Praxis und Forschung

6.3 Ausblick

7 Literaturverzeichnis

ANDERSON, C., 2015. Docker [software engineering]. *Ieee Software*. Jg. 32, Nr. 3, S. 102–c3.

BLOCH, J., 2006. How to design a good API and why it matters. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, S. 506–507.

CHAKRABORTY, M.; KUNDAN, A.P., 2021. Grafana. In: *Monitoring Cloud-Native Applications*. Springer, S. 187–240.

DOGLIO, F., 2015. *Pro REST API Development with Node. js*. Apress.

DOSSOT, D., 2014. *RabbitMQ essentials*. Packt Publishing Ltd.

DUBOIS, P., 2008. *MySQL*. Pearson Education.

FIELDING, R.; GETTYS, J.; MOGUL, J.; FRYSTYK, H.; MASINTER, L.; LEACH, P.; BERNERS-LEE, T., 1999. *RFC2616: Hypertext Transfer Protocol–HTTP/1.1*. RFC Editor.

FIELDING, R.T., 2000. *Architectural Styles and the Design of Network-based Software Architectures – Dissertation* [online]. [besucht am 2022-12-21]. Abger. unter: `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf`.

FRAUCHIGER, D., 2017. Anwendungen von Design Science Research in der Praxis. In: *Wirtschaftsinformatik in Theorie und Praxis*. Wiesbaden: Springer Fachmedien Wiesbaden, S. 107–118.

GÖRZ, G.; SCHNEEBERGER, J., 2010. *Handbuch der künstlichen Intelligenz*. Walter de Gruyter.

GRINBERG, M., 2018. *Flask web development: developing web applications with python*. O'Reilly Media, Inc.

IONESCU, V.M., 2015. The analysis of the performance of RabbitMQ and ActiveMQ. In: *2015 14th RoEduNet International Conference-Networking in Education and Research (RoEduNet NER)*. IEEE, S. 132–137.

JONES, M.; BRADLEY, J.; SAKIMURA, N., 2015. *Json web token (jwt)*. Techn. Ber.

JOSHEPH, T., 2021. Python. *Python Releases for Windows*. Jg. 24.

MASSE, M., 2011. *REST API design rulebook: designing consistent RESTful web service interfaces*. O'Reilly Media, Inc.

MDN, 2022. *HTTP request methods* [online]. [besucht am 2022-12-19]. Abger. unter: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`.

MOISEEV, A.; FAIN, Y., 2018. *Angular Development with TypeScript*. Simon und Schuster.

PAKSULA, M., 2010. Persisting objects in redis key-value database. *University of Helsinki, Department of Computer Science*. Jg. 27.

RAHUTOMO, F.; KITASUKA, T.; ARITSUGI, M., 2012. Semantic cosine similarity. In: *The 7th international student conference on advanced science and technology ICAST*. Bd. 4, S. 1. Nr. 1.

RICHARDS, R., 2006. Representational state transfer (rest). In: *Pro PHP XML and web services*. Springer, S. 633–672.

STACKOVERFLOW, 2022. *How do I ask a good question?* [online]. [besucht am 2022-12-15]. Abger. unter: `https://stackoverflow.com/help/how-to-ask`.