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:	4. Januar 2023

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
GUI Graphical User Interface
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
ID Identifikation
IEC International Electrotechnical Commission
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
WSL Windows-Subsystem für Linux

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

Eine Schnittstelle in der Informatik beschreibt sowohl den Bereich einer Software, der nach außen hin für andere Programme oder den Nutzer erreichbar ist, als auch eine eigenständig Anwendunge, die zwei unabhängige Programme miteinander verbindet. Innerhalb einer Schnittstelle sind Funktionen vorgegeben, mit denen der Nutzer mit der Software interagieren kann. Oftmals werden Schnittstellen für eine spezifische Anwendung entwickelt, um die Nutzung dieser Anwendung zu ermöglichen. Das Ziel einer Schnittstelle ist es, die Komplexität von bestehenden Programmen zu verringern. Der Satz der bereitgestellten Funktionen dient zur Vereinfachung der Bedienung komplizierter oder alter Programme.[1]

Innerhalb des Prototyps für die Bachelorarbeit wurden zwei Schnittstellen entworfen. Die erste Schnittstelle innerhalb der Softwarearchitektur besteht zwischen der Website und dem Backend. Innerhalb des Backends wurde eine Schnittstelle in Form eines Application Programming Interface (API) implementiert. Die zweite implementierte Schnittstelle ist die Verbindung zwischen dem Backend und den einzelnen KI-Services. Zu diesem Zweck wurde RabbitMQ verwendet. Im folgenden Kapitel wird der Aufbau und die Funktionsweise der beiden Schnittstellen genauer erläutert.

2.2 REST und RabbitMQ

2.2.1 Application Programming Interface

Eine 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.[2] 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 eine 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 den Anwender von sich aus nicht ansprechen.

[1] Sneed, 2006.
[2] Bloch, 2006.

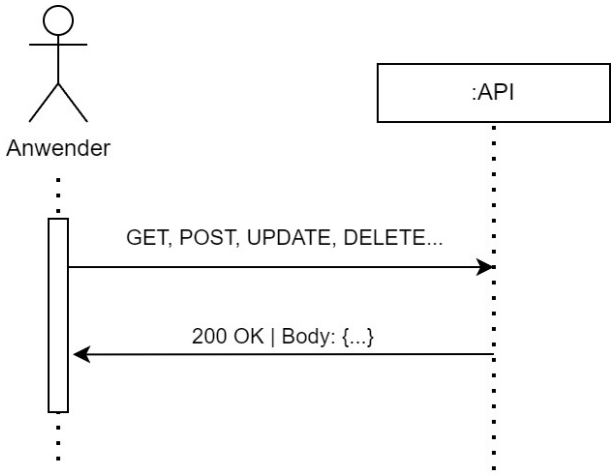

Abbildung 1: UML Sequenzdiagramm einer API

Der gesamte Satz der verfügbaren HTTP-Methoden lautet `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `CONNECT`, `OPTIONS`, `TRACE`, `PATCH`. Jede der Methoden hat eine bestimmte Aufgabe und bestimmte Rechte, die es vom Entwickler der API einzuhalten gilt.[3] 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 auf dem Server vorzunehmen. Wenn zweimal hintereinander die gleiche URL mit `GET` aufgerufen wird, sollte auch in beiden Fällen das gleiche Ergebnis von der API zurückgegeben werden.

Die `POST` Methode liefert Daten vom Nutzer an den Server. Innerhalb des `POST` Bodys können Daten im JSON Format an die API gesendet werden. Neben dem JSON Format gibt es auch noch weitere Möglichkeiten für Formate. Diese werden in der Bachelorarbeit jedoch nicht behandelt. 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 andere Methoden eventuell davon beeinflusst werden. Ein beispielhafter Anwendungsfall für eine `POST` Methode ist die Registrierung eines neuen Accounts auf einer Website. Dort werden 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 eine Fehlermeldung 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.

[3] MDN, 2022.

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 Prototyp 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 eine fehlerhafte 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.[4]

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.[5]

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.

[4] Masse, 2011.
[5] Richards, 2006.

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 Caching 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.[6]

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.[7]

Bei einer grundlegenden Kommunikation gibt es zwei Kommunikationspartner, bei der einer den Sender darstellt und der andere den 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 2 visualisiert.

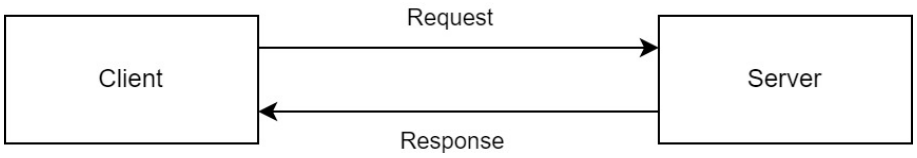

Abbildung 2: Grundlegende Kommunikation

Durch die Art der synchronen Kommunikation sind Client und Server sehr eng miteinander verbunden. Der Client erwartet eine Response von genau dem Server, an den er auch den

[6] R.T. Fielding, 2000.
[7] Dossot, 2014.

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 das 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 zurückkam. Problematisch wird diese Herangehensweise, wenn die Antwortzeit sehr lang wird oder ungewiss ist, ob überhaupt eine Antwort kommen wird. Des Weiteren kann es zu Problemen führen, wenn sehr viele Nachrichten zur gleichen Zeit beim Server eintreffen. Der Server versucht alle eingehenden Nachrichten gleichzeitig anzunehmen und zu verarbeiten. Dadurch entsteht eine sehr hohe temporäre Last. Sollte der Server seine Ressourcen, wie CPU oder RAM nicht dynamisch hochskalieren können, führen die eingehenden Anfragen zu einer Überlastung und damit zum Absturz des gesamten Servers.[8] Um dies zu verhindern, muss ein Zwischendienst implementiert werden, der die eingehenden Nachrichten annimmt, zwischenspeichert und zum passenden Zeitpunkt weiterleitet, wenn der Server die erforderlichen Kapazitäten zum Verarbeiten einer neuen Anfrage hat.

RabbitMQ ist ein eine im Jahr 2007 veröffentlichte, nachrichtenorientierte Middleware, die eine Kommunikation zwischen zwei oder mehreren Programmen durch das Advanced Message Queuing Protocol (AMQP) ermöglicht. RabbitMQ implementiert einen 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.[9] In Abbildung 3 ist die Kommunikation zwischen Client und Server mittels eines Brokers abgebildet.

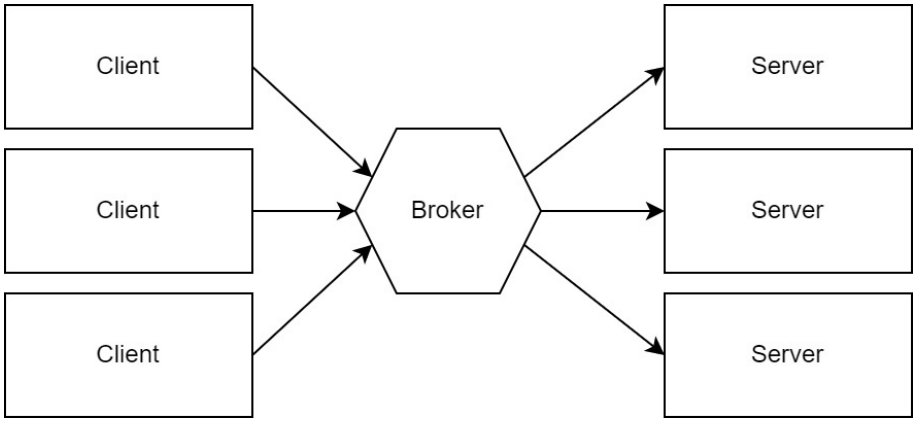

Abbildung 3: Kommunikation über einen Broker nach Dossot

Durch diese Herangehensweise wird eine asynchrone Kommunikation zwischen Client und Server ermöglicht. Sollte ein Server oder ein Client während seiner Laufzeit abstür-

[8] Hoque u. a., 2015.
[9] Johansson u. a., 2020.

zen, hat dies keine Auswirkungen auf die anderen Instanzen. Des Weiteren kann die Anzahl der Clients und Server, abhängig von der aktuellen Nutzlast, dynamisch hoch- und runterskaliert werden. Eine weitere Eigenschaft, die durch diese Architektur entsteht, ist, dass der Client keine Information darüber besitzen muss, wo sich der Server befindet, oder mit welcher Technologie er implementiert wurde. Solange der Server die Schnittstelle zum Broker implementiert, können die Nachrichten vom Client empfangen werden.[10]

Das AMQP ist ein offener Standard, der den Nachrichtenaustausch zwischen Produzent und Konsument regelt. AMQP definiert den gesamten Ablauf des Austausches. Innerhalb des Protokolls wird definiert, wie die Transmission Control Protocol (TCP) Verbindung zwischen Client und Broker aufgebaut wird. Ebenfalls Bestandteil des Protokolls ist der softwareseitige Aufbau der Nachrichtenübermittlung. AMQP implementiert mehrere Konzepte, die auch im Prototyp verwendet wurden. Die folgende Aufteilung der Konzepte wurde durch Johansson und Dossot beschrieben.[11]

Der Broker ist das grundlegende Konzept von AMQP. Er empfängt Nachrichten von einer Anwendung und leitet sie an eine andere Anwendung weiter.

Der Virtual Host, welcher auch vhost genannt wird, bietet eine Möglichkeit mit mehreren Anwendungen auf dem gleichen Broker zu arbeiten, jedoch ohne dass die Anwendungen Einfluss aufeinander nehmen können. Der vhost bietet eine logische Trennung der Anwendungen innerhalb der gleichen RabbitMQ Instanz.

Eine Connection beschreibt die physische Verbindung zwischen einer Anwendung und dem Broker. Die Verbindung zwischen Anwendung und Broker wird mittels TCP hergestellt.

Ein Channel ist eine virtuelle Verbindung innerhalb einer Connection. Wenn ein Client eine Nachricht an einen Broker sendet, wird dafür eine virtuelle Connection aufgebaut, statt jedes Mal eine neue TCP Verbindung zu nutzen. Innerhalb einer Connection können mehrere Channels aufgebaut und genutzt werden. Dadurch können auch mehrere Anfragen über die gleiche TCP Verbindung abgearbeitet werden.

Ein Exchange hat die Aufgabe, die Nachrichten zwischen Anwendung und Broker zu vermitteln. Der Exchange stellt sicher, dass die Nachrichten ankommen und in die richtige Queue geschrieben werden. In welcher Queue die Nachrichten landen, ist abhängig von den Regeln, die durch den Exchange Typen definiert werden.

Eine Queue ist eine Datenstruktur, in der mehrere Operationen definiert werden. Die Queue fungiert als Warteschlange, in der Einträge gespeichert und in der Reihenfolge des Eingangs auch wieder ausgelesen werden. 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.

[10] Dossot, 2014.
[11] Johansson u. a., 2020.

Binding beschreibt eine virtuelle Verbindung zwischen einem Exchange und einer Queue innerhalb des Brokers. Diese Verbindung ermöglicht Nachrichtenfluss von einem Exchange zu einer Queue.

In Abbildung 4 sind die Konzepte von AMQP und deren Ablauf visualisiert.

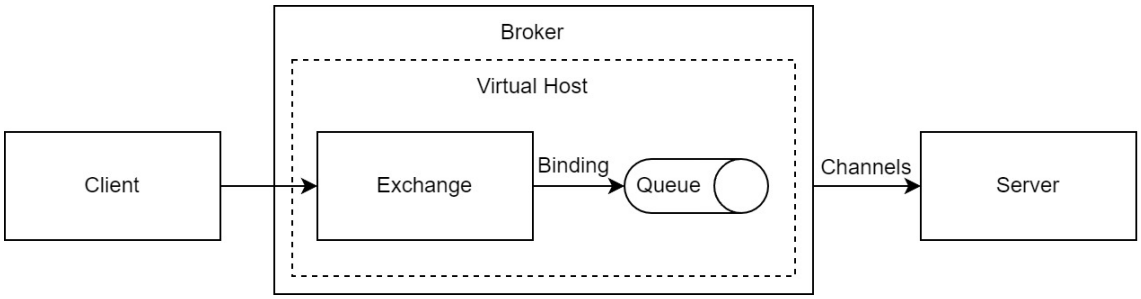

Abbildung 4: Darstellung der AMQP Konzepte nach Dossot

Die Middleware RabbitMQ wird im Prototyp für die Kommunikation zwischen der Flask API und den KI-Services genutzt. Sowohl das Backend, in dem die API implementiert ist, als auch die einzelnen KI-Serives sind als Teil einer Microservice Architektur implementiert. RabbitMQ wird als universelle Schnittstelle für die Kommunikation in der Microservice Architektur verwendet.

2.3 Microservice Architekturen

Das Konzept von Microservices beschreibt die Unterteilung einer umfangreichen Anwendung in einzelne Teilanwendungen. Eine Teilanwendung wird dabei Microservice genannt. Die einzelnen Teilanwendungen sollen dabei untereinander kommunizieren können, um die Zusammenarbeit zu gewährleisten. Dazu nutzen die Teilanwendungen eine universelle Schnittstelle. Eine Anwendung soll nur eine Aufgabe erledigen. Wie umfangreich diese Aufgabe ist, ist jedoch nicht genauer definiert.[12]

Die Idee von Microservices ist es, ein großes System in viele kleine einzelne Systeme aufzuteilen. Dadurch entsteht eine Modularisierung der einzelnen Services. Voneinander unabhängige Softwaremodule können im Gegensatz zu einem Softwaremonolithen unabhängig voneinander deployed werden. Dies ermöglicht es zur Laufzeit bestimmte Teile des Softwaresystems hoch- und runterzufahren. Ein Austausch von fehlerhaften Services oder das Dazuschalten von neuen Services bei mangelnder Leistung ist, während das Softwaresystem in einer Produktivumgebung genutzt wird, möglich.[13]

Jedes Teilsystem kann einzeln deployed werden. Dadurch kann die Verfügbarkeit des gesamten Systems gewährleistet werden, auch wenn bestimmte Bereiche der Software aktualisiert oder verändert werden müssen. Bei einem Monolithen müsste für jede Änderung am System die gesamte Anwendung heruntergefahren und neugestartet werden.

Microservices sind voneinander unabhängige Programme und damit auch eigenständige Prozesse. Die Technologie, mit der ein Service implementiert wird, ist nicht durch die

[12] Wolff, 2018.
[13] Newman, 2015.

allgemeine Architektur oder das System vorgegeben. Jeder Service kann in der für ihn passenden Technologie und Programmiersprache entwickelt werden.

Da die Technologien der Services unabhängig voneinander sind, muss es eine gemeinsame Schnittstelle zur Kommunikation geben. Über diese Schnittstelle werden Daten und Information ausgetauscht, die die jeweiligen Services zum Bearbeiten ihrer Aufgaben benötigen. Die Kommunikation im Prototyp wird mit RabbitMQ implementiert.

2.4 Künstliche Intelligenz

Der Begriff Künstliche Intelligenz (KI) beschreibt eine Software, die das Ziel hat, typisch menschliche Aufgaben, wie das Sehen, Hören und Verstehen von Kontext abzubilden.[14] Künstliche Intelligenzen benötigen eine große Menge an Daten als Grundlage, mit denen sie trainiert werden können. Ein nicht trainiertes KI-Modell kann nicht viel mehr als zufällige Entscheidungen treffen. Erst durch das Training bekommen die Entscheidungen der KI eine Struktur. Nachdem eine KI für ihren bestimmten Einsatzzweck mit den entsprechenden Daten trainiert wurde, kann sie verwendet werden. Die Verwendung einer KI folgt den Grundprinzipien eines normalen Algorithmus. Es wird ein Input angenommen, wie beispielsweise ein Text oder ein Bild. Dieser wird durch die KI analysiert und in eine für die KI verständliche Form gebracht. Diese Form ist im Falle des Prototyps ein 512-Dimensionaler Vektor. Anschließend generiert die KI einen Output. Dieser kann das Endergebnis der Anfrage sein oder für weitere Verarbeitungsschritte genutzt werden.[15]

Künstliche Intelligenzen können von Grund auf selbst entwickelt und trainiert werden. Die Anwendungszwecke einer KI überschneiden sich jedoch oftmals mit bereits vorhandenen Modellen, die es frei oder gegen eine Nutzungsgebühr auf dem Markt gibt. Wenn eine fertig trainierte KI genutzt werden kann, spart dies Entwicklungszeit und Ressourcen. Die vortrainierten KIs werden jedoch meist als Blackbox angeboten. Eine Blackbox beschreibt einen geschlossenen Raum, der eine Eingabe annimmt und eine Ausgabe produziert, ohne dass der Nutzer der Blackbox sehen kann, wie der Algorithmus funktioniert, der zu der Ausgabe führt.

Ein KI-Service beschreibt dabei eine Implementierung einer solchen Blackbox. Ein KI-Service ist ein alleinstehendes Programm, das die Aufgabe hat, 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 Schnittstelle implementiert sein, über die die Nachrichten empfangen werden können. Innerhalb des Prototyps wird in den einzelnen KI-Services RabbitMQ als Schnittstelle genutzt.

Im Prototyp zur Anbindung von austauschbaren Datenquellen an KI-Algorithmen wurde ein Service zur Textähnlichkeitssuche implementiert. Dieser nutzt das Bidirectional Encoder Representations from Transformers (BERT) Modell von Google. Dieser Service

[14] Görz u. a., 2010.
[15] Hamet u. a., 2017.

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 Prototyp 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 Vektors 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.[16] Diese Abstandsberechnung kann in der Elasticsearch Datenbank seit Version 7.3, welche am 31.07.2019 veröffentlicht wurde, automatisch durchgeführt werden.

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.[17]

Flask ist eine der verfügbaren Bibliotheken, die ein Framework für die Implementierung einer webbasierten API bereitstellt. Eine API dient dazu, Funktionen und Routen zu defi-

[16] Rahutomo u. a., 2012.
[17] Josheph, 2021.

nieren, 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.[18]

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 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 gewährleistet. Für den Einsatzzweck als Cache in einer Cloud Umgebung ist Redis allerdings sehr gut geeignet.[19]

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.[20]

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,

[18] Grinberg, 2018.
[19] Paksula, 2010.
[20] DuBois, 2008.

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.[21]

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 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.[22]

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 Prototyp wurde eine MySQL verwendet, in der die zu loggende Nachricht, der Schweregrad, ein Zeitstempel und die User Identifikation (ID) gespeichert wer-

[21] Moiseev u. a., 2018.
[22] Chakraborty u. a., 2021.

den. 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.[23]

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

[23] Anderson, 2015.

3 Methodik

text

3.1 Wasserfallmodell

text[24]

3.2 Evaluationsmethode

text

[24] 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. Zunächst werden die Anforderungen für das Projekt innerhalb der Bachelorarbeit erläutert. Anschließend wird das erarbeitete Konzept beschrieben. Der mithilfe des Konzeptes entwickelte Prototyp sowie das anschließende Deployment findet sich ebenfalls in diesem Kapitel wieder.

4.1 Anforderungen

Die Anforderungen für die Schnittstelle und den daraus resultierenden Prototyp wurden in Zusammenarbeit mit der CONET Solutions GmbH erhoben. Bei der CONET gab es ein offenes Forschung- und Entwicklungsprojekt zur Erstellung eines Trend-Scouting Systems. Das Ziel des Systems sollte es sein, eine Architektur bereitzustellen, die vom Nutzer gestellte Anfragen annimmt und diese mit einer KI analysiert, um Aussagen darüber treffen zu können, welche Technologien in der Zukunft relevant werden könnten. Da der Umfang eines Trend-Scouting Systems zu groß für den Zeitraum einer Bachelorarbeit ist, ergab sich als Zielsetzung die Entwicklung der Architektur, die ein solches System unterstützen könnte. Die genaue Anforderungserhebung wurde nach dem Leitfaden der von der International Electrotechnical Commission (IEC) veröffentlichten Norm IEC 62304 Kapitel 5.2 entworfen.[25] Innerhalb dieser Norm ist spezifiziert, wie Anforderungen an eine Software aufbereitet und dokumentiert werden sollen.

Im ersten Schritt wurden die Benutzerschnittstellen spezifiziert. Die grundlegende Interaktion mit dem System findet über die Website statt. Dort hat der Nutzer die Möglichkeit seine Suchanfrage in Form eines ausgeschriebenen Satzes oder über einzelne Stichpunkte in ein Textfeld einzugeben. Die nächste Benutzerschnittstelle ist die Transformation der zuvor eingegebenen Suchanfrage. Der Nutzer soll dort mittels JSON-Syntax die Suchanfrage modifizieren können. Dies ist im Hinblick auf eine eventuell automatisierte Nutzung des Systems erforderlich. Im dritten Schritt soll der Nutzer die Option haben, die Suchparameter für die KI anzupassen. Vor allem relevant ist die Anzahl der Ergebnisse, die bei einer gestellten Suchanfrage zurückgegeben werden. An dieser Stelle soll aber auch weitere Parameter eingegeben werden können. Anschließend ist eine Möglichkeit vorgesehen die Anfrage inklusive der Parameter abzuschicken. In Abbildung 5 ist die Interaktion mit dem System in einem Use Case Diagramm dargestellt.

[25] Daniel, 2018.

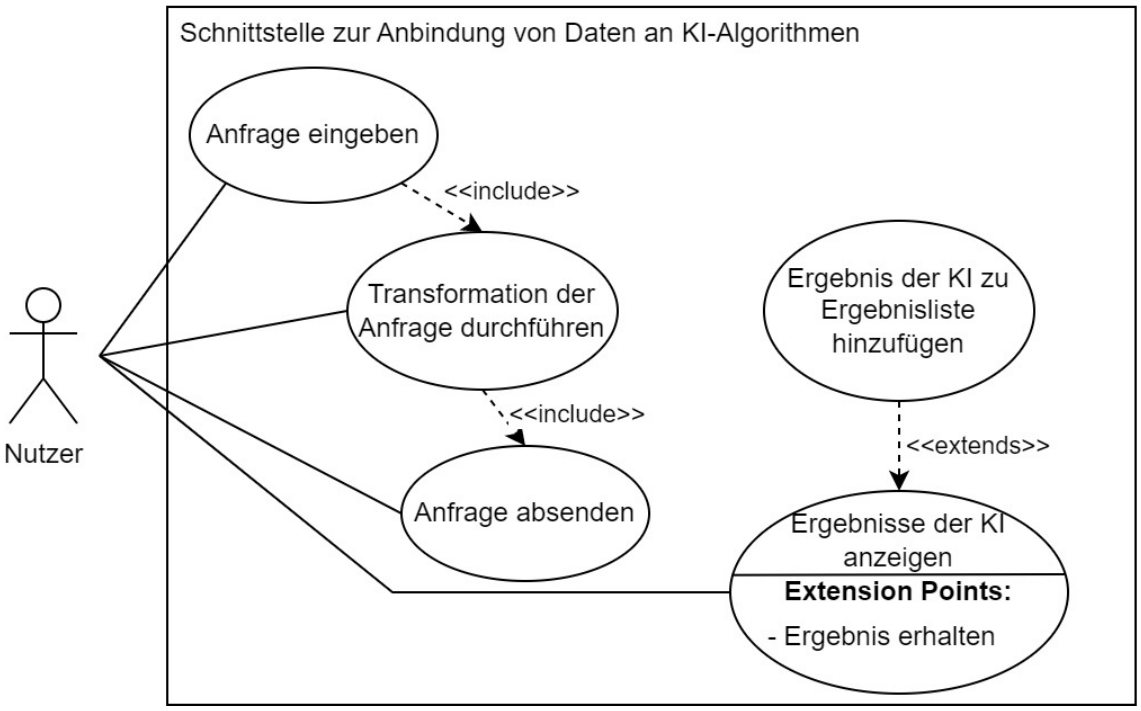

Abbildung 5: Use Case Diagramm der Interaktion mit der Schnittstelle

Die letzte Möglichkeit, die der Nutzer hat, mit der Schnittstelle zu interagieren, ist über die API selbst. Dort gibt es eine Route, über die die aktiven KI-Services verwaltet werden können. Abhängig von der gewählten HTTP-Methode sollen verschiedene Funktionen angeboten werden. Die drei Hauptfunktionen sind die Auflistung aller aktiven Services, das Erstellen eines neuen Services und das Löschen eines Services.

Der nächste Schritt der Anforderungsanalyse nach IEC 62304 ist die Festlegung des Graphical User Interface (GUI). Die grafische Oberfläche sollte sich auf eine Seite beschränken und in drei Bereiche eingeteilt sein. Jeder der Bereiche beinhaltet eine Kachel mit einer Überschrift. Die drei Überschriften „Query“, „Transformation“ und „Service“ sollen verdeutlichen, welche Aktionen in der entsprechenden Kachel passieren. Der linke Bereich der Seite für die Query beinhaltet die Eingabe durch den Nutzer, welche durch ein Textfeld realisiert wird. Des Weiteren gibt es einen Button zum Hochladen der Anfrage an den Server.

Im mittleren Bereich der Seite befindet sich die Kachel für die Transformation der Eingabe. Der Inhalt der Kachel selber ist nochmal in drei Bereiche aufgeteilt. Der erste Bereich zeigt die vom Nutzer hochgeladene Eingabe. In diesem Feld kann der Nutzer keine Änderungen vornehmen. Das zweite Feld enthält die Eingabe für die Transformation. Das Textfeld enthält einen vorgefertigten Text in JSON-Syntax, den der Nutzer anpassen kann. Unter dem Textfeld gibt es einen Button, der die Funktion hat, die Transformationsanleitung an den Server zu senden. Im dritten Feld wird das Ergebnis der Transformation angezeigt, damit der Nutzer überprüfen kann, ob das erzeugte Ergebnis seinen Erwartungen entspricht.

Der rechte Bereich der Seite ist für die Auswahl und Nutzung der KI-Services vorgesehen. Der gewünschte Service soll über ein Dropdown ausgewählt werden. Da die Services für ihre Anfragen Parameter übergeben bekommen müssen, muss eine Tabellenstruktur implementiert werden. Die Tabelle hat zwei Spalten und beliebig viele Reihen. Innerhalb der Tabelle können Schlüssel-Wert Paare eingegeben werden, wobei der Schlüssel in der ersten Spalte und der Wert in der zweiten Spalte steht. Unter der Tabelle muss es einen Button geben, der die Anfrage an den KI-Service sendet. Daraufhin startet der Abfrageprozess der die Antworten der KI-Services zusammenträgt. Die Ergebnisse des KI-Services werden in einem Feld unter dem Button angezeigt. Bei jedem Ergebnis wird zunächst nur die Überschrift angezeigt. Erst bei einem Klick auf die Überschrift soll der Inhalt ausgeklappt werden.

Der Schritt nach der Anforderungserhebung der grafischen Oberfläche ist die Dokumentation des Verhaltens des Systems. Besonderer Fokus liegt hierbei auf dem Verhalten gegenüber Nutzerinteraktionen. Beim ersten Besuch der Website, wird dem Nutzer ein Token zugesendet, welches für alle weitere Anfragen zur Authentifizierung an der API genutzt werden soll. Innerhalb des Tokens ist eine eindeutige User-ID hinterlegt. Nachdem der Nutzer auf der linken Seite der Website in der Kachel „Query“ den Upload Button gedrückt hat, sendet die Website den eingegebenen Text an die API. Dabei soll im Authorization Header der Token stehen und im Request Body der eingegebene Text in JSON-Syntax.

Ist die Anfrage beim Server eingegangen, soll sie in einem Redis Cache hinterlegt werden und anschließend wieder ans Frontend zur Bestätigung zurückgesendet werden. Sollte bei der Anfrage ein Fehler aufgetreten sein, wird der Statuscode 500 an die Website zurückgegeben und das Ergebnis des Uploads wird nicht im ersten Feld der Transformation angezeigt. Sollte der Upload erfolgreich gewesen sein, wird der Text dort angezeigt und der Nutzer hat die Möglichkeit, eine Transformationsanleitung in JSON-Syntax einzugeben und diese über den „Transform“ Button an das Backend zur Auswertung zu schicken. Die Transformationsanleitung wird, nachdem sie im Backend eingetroffen ist, ebenfalls im Redis Cache gespeichert. Das Backend führt eine Transformation mit der gegebenen Anleitung durch und sendet sie zurück an die Website. Das Ergebnis der Transformation wird im dritten Feld der „Transformation“ Kachel angezeigt. Bei einem Fehler während der Transformation wird der Statuscode 500 zurückgesendet und es werden keine Ergebnisse angezeigt.

In der „Service“ Kachel im rechten Bereich der Seite kann der Nutzer über ein Dropdown einen von allen aktiven Services auswählen. Die Liste der Services wird beim Laden der Seite von der API angefragt. Die API stellt dafür eine SQL Anfrage an eine MySQL Datenbank, in der alle aktiven Services hinterlegt sind. Anschließend kann der Nutzer einen oder mehrere Parameter in einer Tabelle eingeben. Der ausgewählte Service und die eingegebenen Parameter werden durch den Klick auf einen „Execute“ Button unterhalb der Tabelle an die API gesendet. Das Backend soll die transformierten Anfragen über RabbitMQ an den dafür vorgesehenen Service schicken. Zur Bestimmung des Services wird der mitgelieferte Servicename genutzt, der durch die Dropdownauswahl ermittelt wird. Je-

de einzelne Anfrage soll eine Anzahl an erwarteten Antworten haben. Das Backend teilt die Gesamtzahl der erwarteten Antworten der Website mit, wo sie zur Anzeige für den Nutzer verwendet werden kann.

Der zu implementierende Service ist eine Textähnlichkeitssuche. Dieser Service erhält die vom Backend gesendeten Anfragen und dazu soll mittels des BERT-Modells die semantisch ähnlichsten Texte aus einer Elasticsearch Datenbank herausfiltern. Die ähnlichsten Texte werden über RabbitMQ wieder an das Backend gesendet, wo sie im Redis Cache zwischengespeichert werden. Die Website soll nach dem Absenden der Anfrage in regelmäßigen Intervallen Anfragen an die API stellen, ob Antworten vom KI-Service im Backend eingetroffen sind. Bei einer solchen Anfrage schaut das Backend im Redis Cache nach, ob eine oder mehrere Antworten existieren. Falls dies der Fall sein sollte, sendet die API die Antworten. Im Fall, dass noch keine Antworten im Redis Cache liegen, gibt das die API eine Antwort mit der Nachricht `successful:False` zurück. Die Website stellt so lange Anfragen an die, bis die erwartete Zahl der Antworten eingetroffen ist.

Der nächste Punkt in der IEC 62304 ist die Festlegung der Geschwindigkeit, in der das System bestimmte Aufgaben abarbeiten soll. Hierzu gab es bezüglich der KI-Services keine genaue Vorgabe seitens der CONET. Das Backend und die Kommunikation zu den einzelnen Service hatte jedoch gewisse Maximallaufzeiten. Eine Anfrage, die durch einen HTTP-Request in der API ankommt, soll maximal 0,5 Sekunden benötigen, um an den richtigen Service weitergeleitet zu werden. Die Rückrichtung vom Service zum Backend soll ebenfalls maximal 0,5 in Anspruch nehmen. Die Gesamtzeit der Kommunikation zwischen Backend und einem Service darf demnach nicht mehr als eine Sekunde betragen. Da der KI-Algorithmus keine festgelegte Laufzeit hat, soll durch die Architektur, mit der die Systeme aufgesetzt sind, kein großer zusätzlicher Overhead entstehen. Antwortzeiten innerhalb einer Sekunde gelten laut der CONET als responsiv und praxistauglich.

In der Spezifikation IEC 62304 wird ebenfalls gefordert, die Umgebung, in der die Software installiert werden soll, zu definieren. Die Hauptanforderung der CONET war hierbei, dass das System möglichst plattformunabhängig entwickelt werden soll. Für die Installation der einzelnen Komponenten ist Docker vorgesehen.

Der letzte festgehaltene Punkt aus der IEC 62304 ist die Beschreibung, wie das System auf Benutzerfehler und Überlast reagieren soll. Grundsätzlich sollen Benutzerfehler im Backend abgefangen werden und nicht zum Serverabsturz führen. Dazu soll ein System implementiert werden, was potentielle Fehler abfängt und anschließend in eine MySQL Datenbank logged. Des Weiteren sollen alle aufgerufenen Routen und Vorgänge innerhalb des Backends, sowie den einzelnen Services geloggt werden. Dadurch kann nachvollzogen werden, ob das System ordnungsgemäß funktioniert. Die Architektur soll horizontal skalierbar sein. Das bedeutet, dass mehrere Instanzen der API und der einzelnen Services parallel betrieben werden können. Jede dieser Instanzen muss für sich allein stehend voll funktionsfähig sein.

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. Laut der CONET sind die drei wichtigsten nicht funktionalen Anforderungen 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 Softwarearchitektur sind in Abbildung 6 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 Prototyp 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 ähnlichsten Texte herausfiltern. Die gefunden Einträge werden über RabbitMQ im Anschluss wieder an das Flask Backend geschickt, damit sie dort vom Frontend ausgelesen werden können.

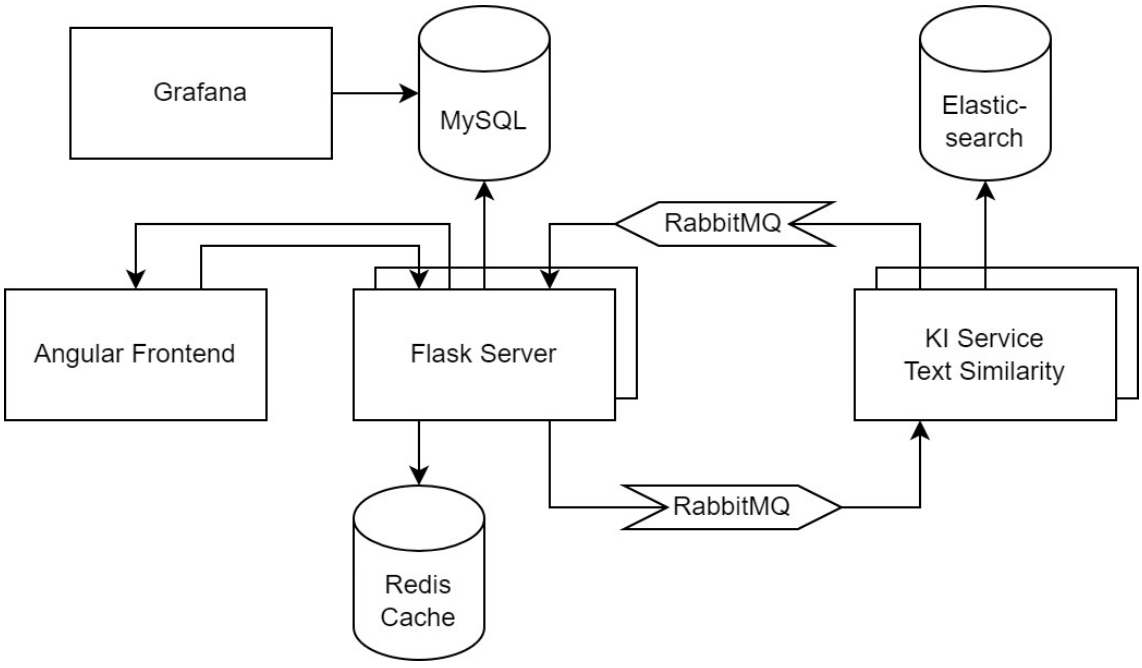

Abbildung 6: Softwarearchitekturdiagramm

4.2.2 Programmablauf

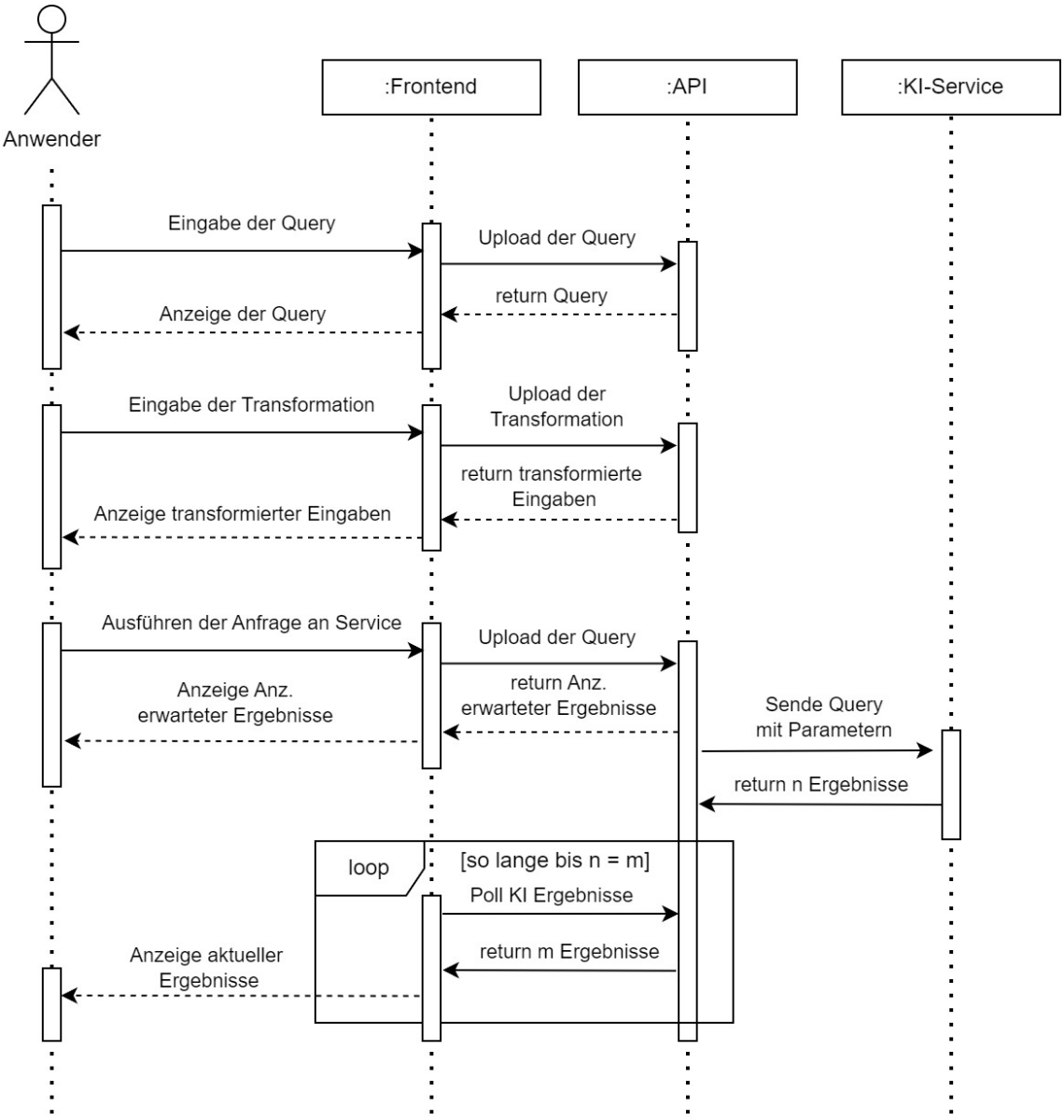

Abbildung 7: UML Sequenzdiagramm des Programmablaufs

Das Ziel der Bachelorarbeit ist es, eine Schnittstelle zu entwickeln, die Daten an KI-Algorithmen anbindet. Dazu wird im Prototyp ein System implementiert, welches die Anbindung in drei Schritten durchführt. In Abbildung 7 ist der grundlegende Programmablauf in einem UML Sequenzdiagramm dargestellt. In Abbildung 8 ist ein Mockup der gesamten Website abgebildet. Alle Interaktionen, die der Anwender auf der Website tätigen kann, sind hier beispielhaft abgebildet. Die Mockups dienen der Veranschaulichung des entwickelten Konzepts für die Nutzerinteraktion mit der Schnittstelle.

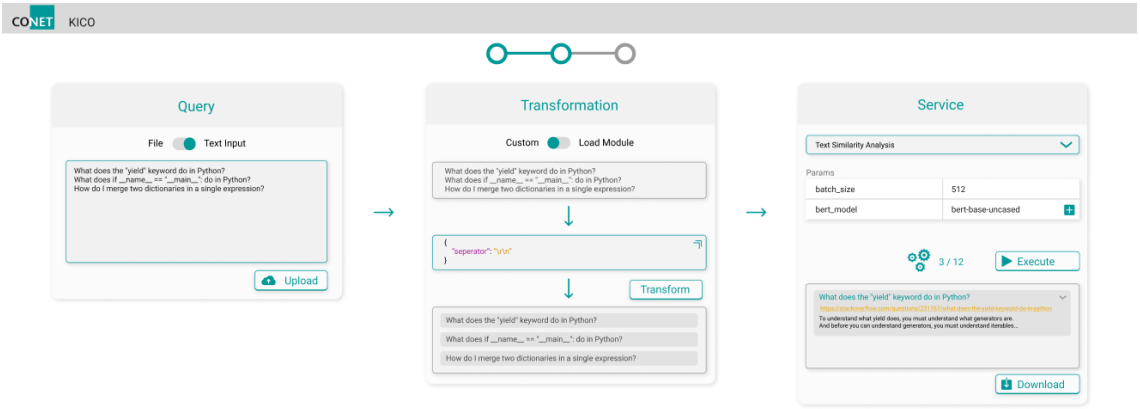

Abbildung 8: Mockup Frontend

Der erste Schritt ist, wie auch in den Anforderungen definiert, die Eingabe der Query durch den Anwender. Auf der Website gibt es dafür ein Textfeld, in dem diese Eingabe stattfindet. Eine Darstellung der „Query“ Kachel ist in Abbildung 9 gegeben. Die Eingabe kann durch einen Klick auf den „Upload“ Button an die API per HTTP POST-Request geschickt werden. Innerhalb des Bodys befindet sich die Eingabe. Zusätzlich zu dem Body wird ein Authorization Header mitgeschickt, der die User-ID des Anwenders enthält. Innerhalb der API wird die User-ID als Schlüssel und die Qeury als Wert für einen Eintrag in den Redis Cache genutzt. Die Eingabe wird im Backend zunächst nur zwischengespeichert. Als Bestätigung für den Nutzer, sendet die API die erhaltene Query wieder an die Website zurück. Dort wird sie im oberen Bereich der „Transformation“Kachel angezeigt.

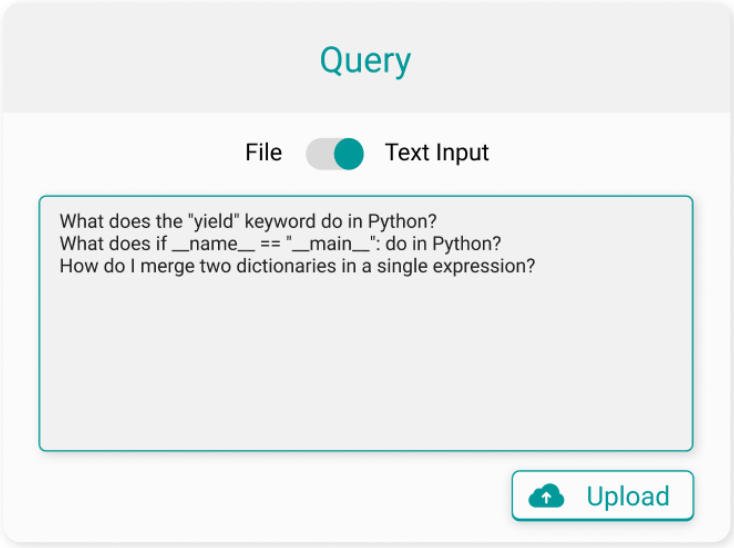

Abbildung 9: Mockup Query

Der zweite Schritt ist die Eingabe der Transformationsanleitung. In Abbildung 10 ist das Mockup für die „Transformation“ Kachel abgebildet. Im mittleren Textfeld der „Transformation“ Kachel ist für diesen Schritt ein Textfeld vorgesehen. Nach Abschluss der Eingabe kann der Anwender auf den Button „Transform“ klicken, um die Eingabe an die API hoch-

zuladen. Wie auch bei dem Upload der Query wird die User-ID im Header des Requests mitgeschickt. Die API nutzt die User-ID um die zuvor hochgeladene Query aus dem Redis Cache zu laden. Die im Request mitgelieferte Transformationsanleitung wird in diesem Schritt genutzt, um die Query zu transformieren. Im Prototyp wird eine Split-Operation beliebige Anzahl an Replace-Operationen implementiert. Die Split-Operation teilt die Query in mehrere einzelne Queries. Dazu wird eine Zeichenkette genutzt, die als Trennzeichen dient. Die Replace-Operation ersetzt eine eingegebene Zeichenkette durch eine andere. Das Ergebnis der Transformation wird innerhalb der Response wieder an die Website gesendet. Sollte eine Split-Operation angegeben worden sein, werden alle daraus resultierenden Queries im unteren Bereich der „Transformation“ Kachel angezeigt.

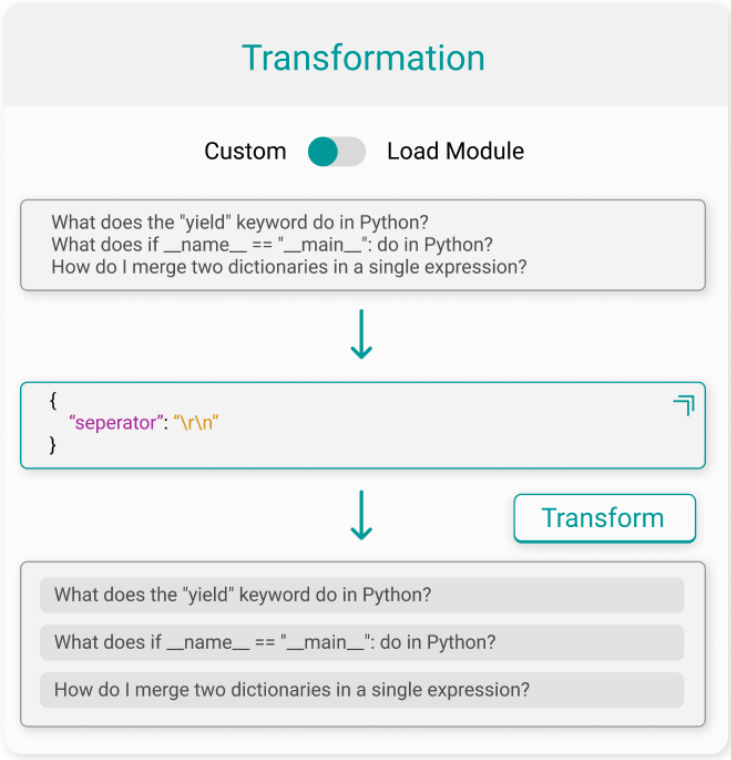

Abbildung 10: Mockup Transformation

Der dritte Schritt ist die Ausführung der Anfrage durch einen KI-Service. In Abbildung 11 ist das Mockup für die „Service“Kachel dargestellt. Der Anwender wählt im Dropdown der Kachel „Service“ einen der in der MySQL Datenbank hinterlegten aktiven Services aus und gibt anschließend einen oder mehrere Parameter in die Tabelle unterhalb des Dropdowns ein. Durch einen Klick auf den Button „Execute“ wird eine Query im Frontend zusammengebaut und als Request an die API gesendet. Innerhalb der Query befindet sich der Name des KI-Services, sowie alle in der Tabelle definierten Parameter. Wenn der Request bei der API eingegangen ist, lädt das Backend sowohl die in Schritt eins eingebene Query, sowie die in Schritt zwei eingegebene Transformationsanleitung. Das Backend führt nun eine erneute Transformation durch und sendet dem Frontend die Anzahl der erwarteten Antworten des KI-Services zurück. Die Anzahl setzt sich aus der Menge der durch die Split-Operation entstandenen Queries und den eingegebenen Parametern zusammen.

Für den KI-Service zur Textähnlichkeitssuche gibt es den Parameter `search_size`. Die Anzahl der erwarteten Antworten wird im Frontend zunächst zur Fortschrittsanzeige genutzt. Der zweite Einsatzzweck der erwarteten Antworten ist innerhalb der darauffolgenden Schleife. Nach dem Upload der Query startet die Website eine Schleife in der bei der API regelmäßig nach neuen Ergebnissen des KI-Services gefragt wird. Der KI-Service bekommt direkt nach dem Upload der Query über RabbitMQ die Anfrage zugesendet. Innerhalb des Services in dem die Textähnlichkeitssuche implementiert wird, wird die eingehende Nachricht angenommen und mithilfe des BERT-Modells in einer Elasticsearch Datenbank nach den semantisch ähnlichsten Einträgen gesucht. Es werden, abhängig von der `search_size`, die ähnlichsten Ergebnisse gesammelt und über RabbitMQ wieder an das Backend gesendet. Im Sequenzdiagramm in Abbildung 7 ist die Anzahl mit der Variable `n` angegeben. Die Ergebnisse werden in einem Redis Cache zwischengespeichert. Wie zuvor erwähnt, läuft parallel zu der Bearbeitung der Services im Frontend eine Schleife, die die Ergebnisse aus der API abfragt. In einer Schleifeniteration werden alle im Redis Cache hinterlegten Ergebnisse aus dem KI-Service abgefragt und an das Frontend gesendet. Die maximale Anzahl der Ergebnisse ist dabei `m`. Die Schleife läuft so lange, bis die Anzahl der erhaltenen Antworten identisch mit Anzahl der erwarteten Antworten ist, also bis $n = m$ ist. Jedes erhaltene Ergebnis wird im Frontend im unteren Bereich der „Service“ Kachel angezeigt. Damit ist der Programmablauf abgeschlossen und der Anwender kann eine neue Query eingeben.

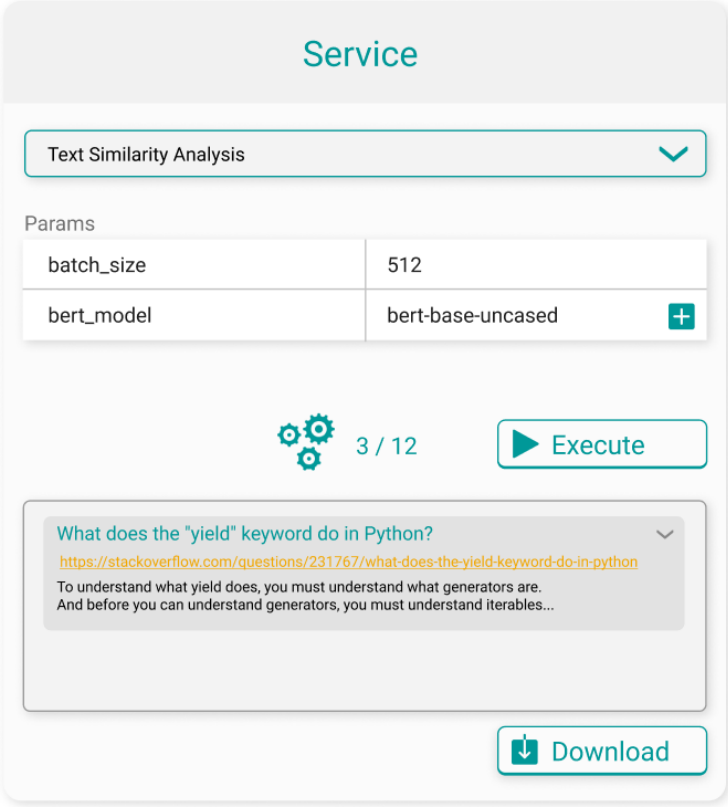

Abbildung 11: Mockup Service

4.3 Prototypische Umsetzung

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, eventu-

elle 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 Header 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.[26]

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 Prototyp 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 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.[27]

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):
    [...]
```

[26] Jones u. a., 2015.
[27] 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 Queries für den Input der KI
2. Datenbank 1: Cache der Transformationsanleitung
3. Datenbank 2: Cache der vom KI-Service produzierten Ergebnisse

Redis wird zum Cachen von Nutzereingaben und für die Zwischenspeicherung von Ergebnissen aus den KI-Services verwendet. In der ersten Datenbank werden die vom Nutzer eingegebenen Queries gespeichert. Redis kann grundsätzlich nur textbasierte Daten speichern. Dies reicht für den Anwendungsfall der Software jedoch aus. Um die Query in Redis speichern zu können, muss zunächst eine Verbindung zum Redis Cache aufgebaut werden. Die URL, unter der Redis erreichbar ist, wird aus den Environmentvariablen bezogen. Im ersten Schritt wird ein `ConnectionPool` angelegt. Dieser verbindet sich unter einem angegebenen Host, einem Port mit einer Datenbank. Anschließend kann im zweiten Schritt eine Redis Instanz in Python erstellt werden. Diese bekommt den `ConnectionPool` als Argument übergeben. Innerhalb der Redis Instanz befinden sich alle Funktionen, die benötigt werden, um mit der Redis Datenbank arbeiten zu können.

```
redis_host = os.environ.get('REDIS_HOST')
pool_upload = redis.ConnectionPool(host=redis_host, port=6379, db=0)
red_upload = redis.Redis(connection_pool=pool_upload)
```

In der Redis Instanz werden zwei Funktionen definiert, die im Prototyp Verwendung finden. Die `set(k,v)` Methode nimmt einen Schlüssel als ersten Parameter und einen Wert als zweiten Parameter an. Das übergebene Key-Value Paar wird mit diesem Methodenaufruf entweder neu in der Datenbank angelegt, oder falls bereits ein identischer Schlüssel vorhanden sein sollte, mit den aktuellen Werten überschrieben. Das Zwischenspeichern der hochgeladenen Query ist im nachfolgenden Codeausschnitt abgebildet.

```
red_upload.set(uid, body['text'])
```

Die zweite Funktion der Redis Instanz ist die `get(k)` Methode. Diese bekommt einen Schlüssel übergeben und gibt den dazugehörigen Wert zurück. Das nachfolgende Codesegment stammt aus der Transform-Route. Hier wird die zwischengespeicherte Query, sowie die Transformationsanleitung aus dem Redis Cache geladen. Wichtig dabei zu beachten ist es, dass die aus dem Redis Cache geladenen Werte noch in ein Zeichenformat konvertiert werden müssen. In diesem Fall ist die Kodierung UTF-8 gewählt worden, da diese die meisten Sonderzeichen unterstützt.

```
messages = transform(red_upload.get(uid).decode('utf-8'), red_transform
    .get(uid))
```

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 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.[28]

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` Methode übergeben. Dieser ist abhängig vom Service, an den die Anfrage gerichtet ist. Im Prototyp 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.

[28] Dossot, 2014.

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 zurückkommt, 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.

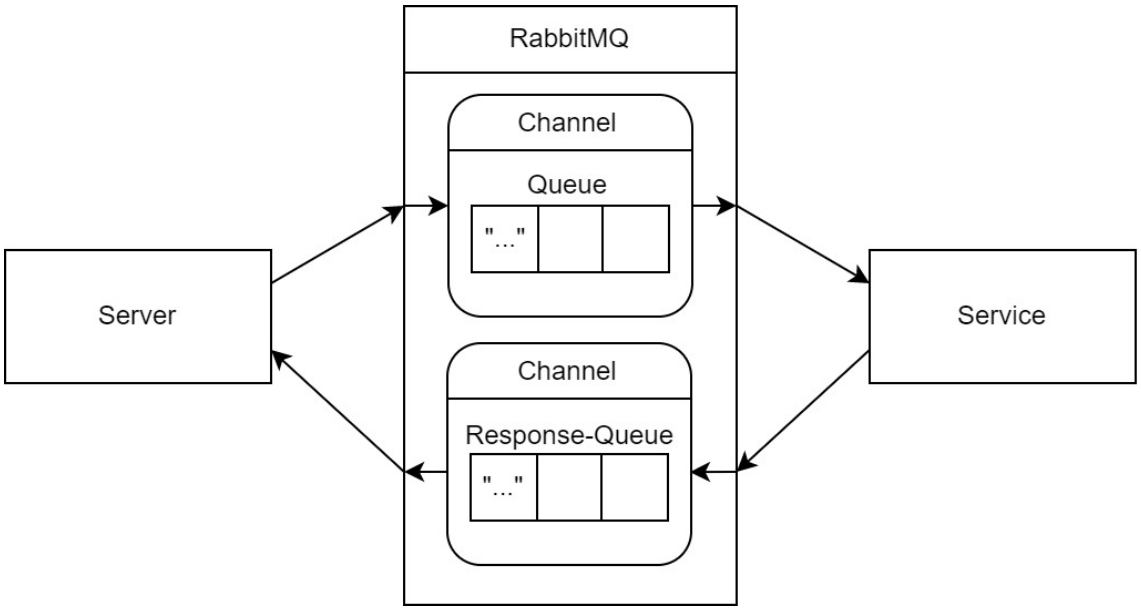

Abbildung 12: Kommunikation mit RabbitMQ

4.3.5 Implementierung des KI-Services

Der implementierte KI-Service lädt im ersten Schritt das KI-Modell, um mit diesem für jeden Eintrag in der Elasticsearch Datenbank einen semantischen Vektor zu generieren. Dieser semantische Vektor wird ebenfalls in der Datenbank gespeichert und kann für zukünftige abfragen genutzt 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 Zu-

stand „Auf Anfrage warten“ über. Veranschaulicht wird der Programmablauf des Services in Abbildung 7.

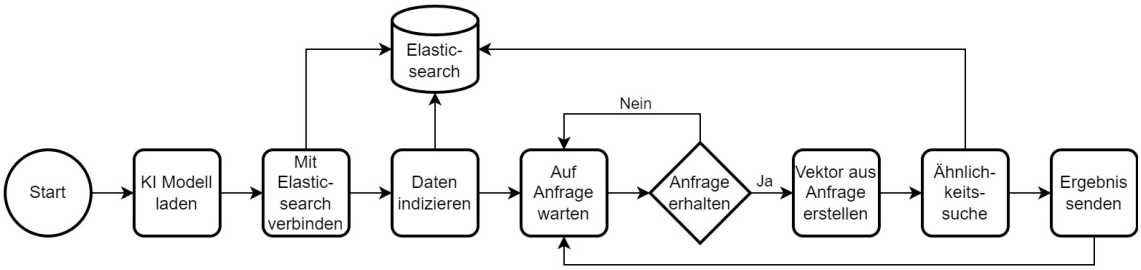

Abbildung 13: Ablaufdiagramm der Textähnlichkeitssuche

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 Prototyp 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.[29]

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

[29] Stackoverflow, 2022.

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"]

    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 Schleifendurchlauf des Services ist damit abgeschlossen. Der Service geht nun wieder, wie in Abbildung 7 zu sehen, zum Zustand „Auf Anfrage warten“ über.

4.3.6 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 mit der Bezeichnung `service` und einen Anzeigenamen mit der Bezeichnung `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.

4.3.7 Automatisierte Transformation des Inputs

Bei der Nutzung eines KI-Services kann es dazu kommen, dass die Eingaben in einer bestimmten Form vorliegen müssen. Dies ist besonders dann von Bedeutung, wenn Künstliche Intelligenzen von Drittanbietern verwendet werden. Da die Implementation oftmals als Blackbox erfolgt und nur eine Schnittstelle nach außen bereitgestellt wird, kann nicht der Algorithmus an die Daten angepasst werden, sondern die Form der Daten muss an den Algorithmus angepasst werden. Daten kommen jedoch häufiger mit überschüssigen Informationen, je nach Quelle und Methode der Datenerhebung. Die Bereinigung per Hand ist sehr aufwändig und führt schnell zu Fehlern. Sollten Echtzeitdaten an einen KI-Service angeschlossen werden, ist eine händische Bereinigung grundsätzlich nicht möglich.

Um diesem Problem entgegenzuwirken, wurde im Backend ein Algorithmus entwickelt, der Daten mithilfe einer Transformationsanleitung automatisch bereinigen kann. Die Transformationsanleitung wird durch den Anwender eingegeben und hat folgende Form:

```
{
  "seperator": "\n",
  "replace": [
    {
      "old":"\[(\d*:?)*\]",
      "new":""
    }
```

```
  ]
}
```

Aktuell im Prototyp unterstützte Transformationsoperationen sind `sperator` und `replace`. Der `sperator` teilt die vom Nutzer eingegebene Query in mehrere einzelne Queries auf. Dazu nutzt er die `split` Operation aus Python, die für Strings standardmäßig verfügbar ist. In der Transformationsanleitung wird ebenfalls ein Array an `replace` Operationen angegeben. Jeder Eintrag im `replace` Array enthält ein Objekt mit Key-Value Paaren `old` und `new`. Der alte String wird durch den neuen String ersetzt. Dazu wird die `replace` Funktion aus dem Python Package `re` verwendet. Die `replace` Funktion nutzt Regular Expression als Syntax für das Ersetzen. Dadurch sind auch komplexere Ersetzungsverfahren möglich. Im Beispiel für eine Transformationsanleitung wurde eine Regular Expression zur Entfernung eines Zeitstempels mit der Form `[12:25:10]` angegeben. Der nachfolgende Code zeigt die Implementierung der `transform` Methode, die mithilfe der Anleitung die Transformation durchführt. Zunächst wird die Query in einzelne Nachrichten geteilt. Anschließend läuft eine Schleife über alle Nachrichten und wendet alle gegebenen Transformationsschritte an. Der Algorithmus gibt ein Array an transformierten Nachrichten zurück.

```
def transform(file: str, transform_json):
    [...]
    parts = file.split(transform_json['seperator'])
    messages = []
    for m in parts:
        for replacer in transform_json['replace']:
            m = re.sub(replacer['old'].replace("\\\\", "\\"), replacer[
                'new'], m)
        m = m.strip()
        messages.append(m)
    return messages
```

4.3.8 Fehlerbehandlung

Während der 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 gespei-

chert. 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.[30]

```
[...]
 @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)
            [...]
```

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.

4.3.9 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 beim Entwickler durch das 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 3 aufgeführt. Die unterstützten Log Level leiten sich aus der Funktionalität von Grafana ab, welche diese Logs mit einer zugeordneten Farbe visualisiert.

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

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. Im folgenden Codeausschnitt ist die Log Funktion abgebildet.

```
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))
       )
    [...]
```

Die produzierten Logs, die in der MySQL Datenbank hinterlegt werden, können in Grafana visualisiert werden. Dazu muss ein Dashboard angelegt werden. Innerhalb des Dashboards wird ein durch Grafana bereitgestelltes Logs Panel benötigt. Grafana ordnet alle anzuzeigenden Daten grundsätzlich nach dem Zeitpunkt der Erstellung. Innerhalb des Logs Panels wird der aktuellste Log ganz oben angezeigt. Alle anderen werden chronologisch absteigend sortiert angezeigt. Da in Grafana die MySQL Datenbank als Datenquelle hinterlegt ist, muss auch die Abfrage der Daten ist SQL stattfinden. Innerhalb der SQL Abfrage ist die Auswahl des timestamps an erster Stelle durch Grafana vorgegeben. Alle weiteren Spalten können beliebig gewählt werden. Wichtig zu beachten ist, dass eine Spalte mit dem Namen „message“ direkt im Logs Panel angezeigt wird, ohne den jeweiligen Log ausklappen zu müssen. Der Name „level“ ist ebenfalls von Grafana reserviert. Sollte sich der Eintrag der „level“ Spalte mit einem der Ausdrücke aus Tabelle 3 decken, so wird die entsprechende Farbe für den Log genutzt. Alle weiteren selektierten Spalten

werden durch Ausklappen des jeweiligen Logeintrags sichtbar. Die genutzte Query zur Abfrage der Logs ist im nachfolgenden Codesegment dargestellt.

```
SELECT timestamp, message, level, uid from db_logs.logs;
```

Das Log Panel mit den aus der Query resultierenden Logs ist in Abbildung 10 abgebildet.

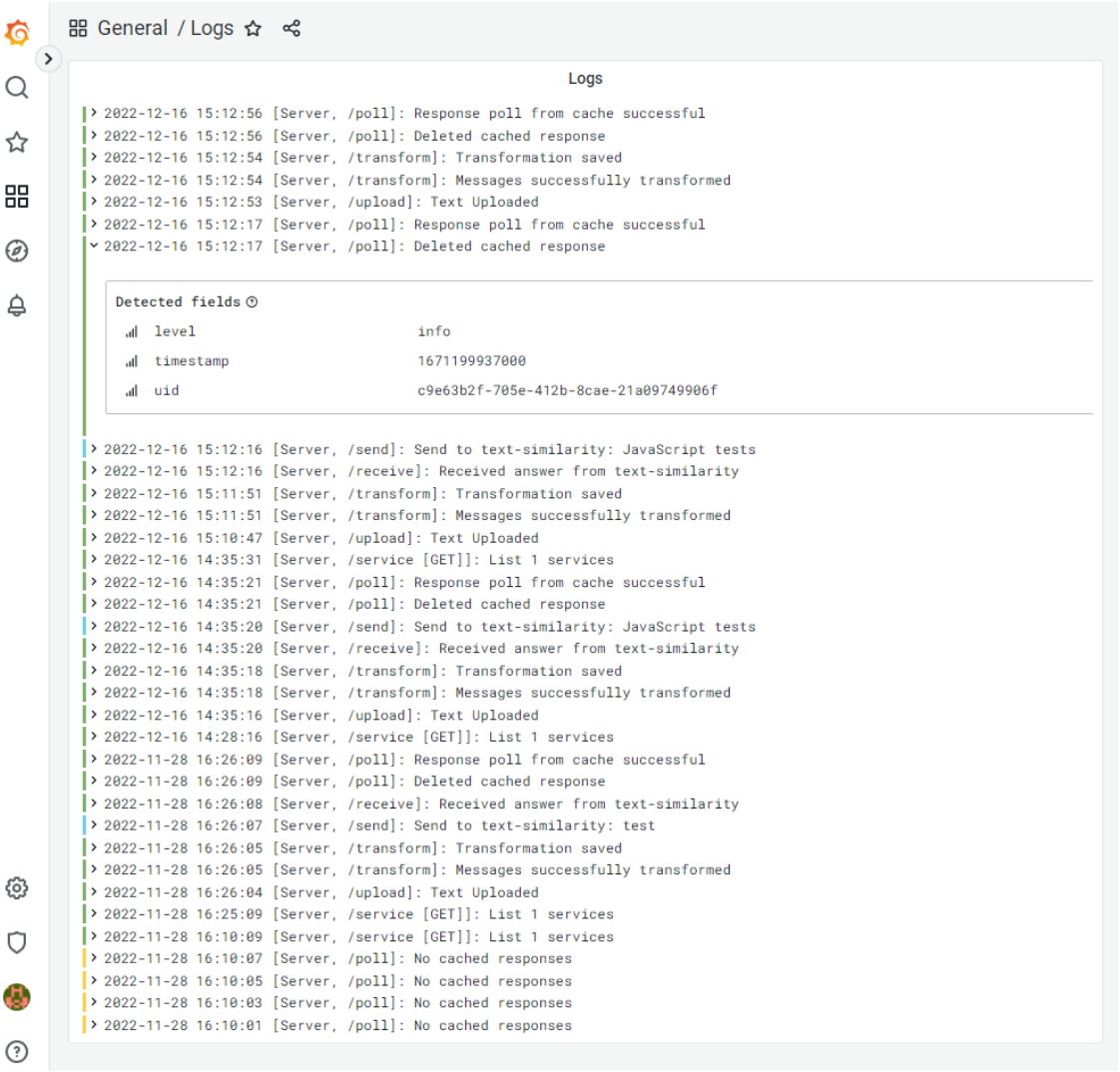

Abbildung 14: Logs in Grafana

4.3.10 Website mit Angular

Damit ein Anwender die Schnittstelle zur Anbindung von Daten an KI-Services nutzen kann, wurde eine Website entworfen und entwickelt. Die grundlegende Funktionsweise der Website ist in den Abschnitten 4.1 Anforderungen und 4.2 Konzeption beschrieben. In diesem Kapitel liegt der Fokus auf der technischen Umsetzung der Website.

Das Frontend wurde mit dem Framework Angular entwickelt. In Abbildung 15 ist der fertiggestellte Prototyp der Website dargestellt.

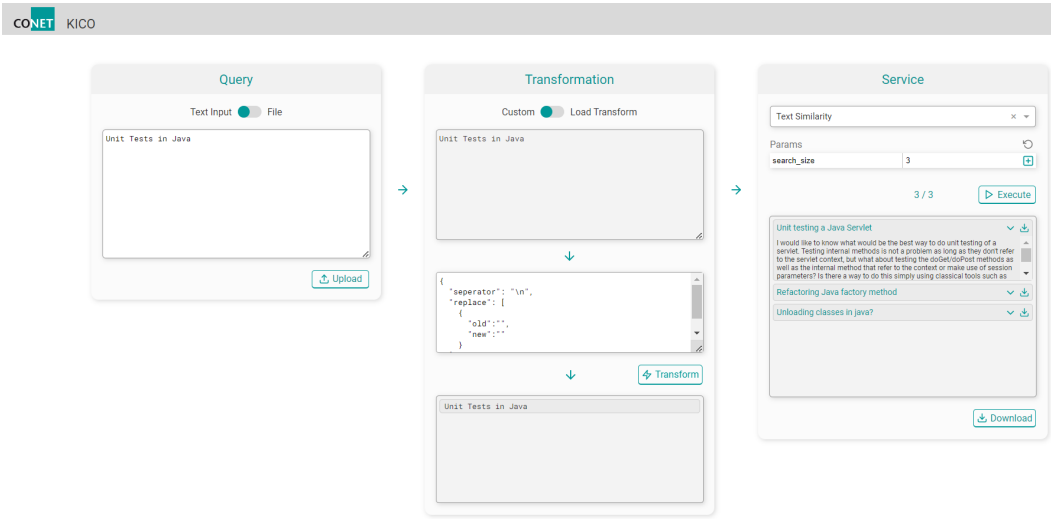

Abbildung 15: Gesamtansicht der Website

Wie in den Anforderungen beschrieben, teilt sich die Website in drei Bereiche auf. In Angular ist es möglich, eigene Components zu definieren. Ein Component besteht aus drei Dateien. Die grundlegende Datei ist eine TypeScript Datei. Innerhalb dieser Datei wird der eigentliche Component definiert. Zunächst muss dafür `Component` aus `@anuglar/core` importiert werden. Im Anschluss im Bereich unter `@Component` ein selector, eine templateUrl und die styleUrls definiert. Der selector beschreibt, wie der Component im HTML Code aufgerufen werden soll. Jeder Component braucht einen individuellen Namen. Um die eigenen Components von den Components, die durch Angular bereitgestellt sind, unterscheiden zu können, befindet sich zu Beginn jedes Namens das Prefix `app-`. Die templateUrl gibt den Pfad der dazugehörigen HTML Datei an. Diese Datei ist die zweite Datei, die zur Erstellung eines Components notwendig ist. Innerhalb der styleUrls können mehrere Pfade zu verschiedenen Dateien angegeben werden. Die SCSS Datei ist die dritte Datei, die zur Erstellung eines Component benötigt wird. Im folgenden Codeausschnitt ist die Initialisierung des Query-Imput-Components innerhalb der TypeScript Datei dargestellt.

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-datasource-input',
  templateUrl: './datasource.input.component.html',
  styleUrls: ['./datasource.input.component.scss']
})
```

Innerhalb der TypeScript Datei können Variablen und Funktionen definiert werden. Diese können dazu genutzt werden, Nutzerinteraktionen mit der Website festzuhalten, oder Ereignisse darzustellen. In der Query-Input-Component wird die Variable `inputStr` definiert. Innerhalb dieser Variable wird die Eingabe, die der Nutzer im Textfeld der Query-Kachel tätigt, gespeichert. Für sich allein stehend, hat die Variable jedoch keine Funktion. Diese bekommt sie erst durch den Code aus der HTML Datei.

```
<textarea [(ngModel)]="inputStr" placeholder="Input..." class="text-
    input"></textarea>
```

Innerhalb der HTML Datei für die Query ist eine Textarea definiert. Über den Selector `[(ngModel)]` kann eine Variable angegeben werden, die mit dem Inhalt der Textarea synchronisiert wird. Sobald der Nutzer eine Eingabe in der Textarea tätigt, wird diese sofort in der TypeScript Datei aktualisiert.

Es ist ebenfalls ein `class` Attribut innerhalb der Textarea angegeben. Dieses setzt eine CSS Klasse für die Textarea. In der Definition der Component ist eine SCSS Datei angegeben, die in diesem Schritt aufgerufen wird. Dort wird die Klasse `text-input` definiert und mit verschiedenen Style-Eigenschaften versehen. Diese Eigenschaften werden auf die Textarea angewandt.

```
.text-input{
  border-radius: 5px;
  border: 1px solid #9A9A9A;
  box-shadow: 2px 2px 8px rgba(0,0,0,.15);
  resize: vertical;
  height: 206px;
  width: 90%;
  font-family: "Roboto Mono", sans-serif;
  padding: 5px;
}
```

Die daraus resultierende Textarea innerhalb der „Query“ Kachel, ist in Abbildung 16 dargestellt.

Abbildung 16: Query Kachel im Frontend

Der Bereich der Transformation besteht aus mehreren Sub-Components. Die obere und mittlere Textarea ist analog zu der Textarea aus der Query implementiert. Da die obe-

re Textarea jedoch ausschließlich zur Bestätigung der Eingabe dient und nicht selbst zu Eingabe genutzt werden soll, wurde bei der Definition das Attribut `readonly` mitgegeben. Der untere Bereich der Transformation-Component teilt sich in zwei Components auf. Der übergeordnete Component ist das `transformation.result`. Dies ist die graue Box, in der eine Liste an transformierten Eingaben dargestellt werden. Jedes Element der Liste ist ein `transformation.result.item` Component. In Angular kann eine Liste im HTML Code direkt aus einer Variable erzeugt werden. Dazu wird `*ngFor` verwendet. Innerhalb der Schleife ist eine Variable aus der zugehörigen TypeScript Datei angegeben, über die iteriert wird. Sollte sich die Variable zur Laufzeit ändern, so wird auch direkt die HTML Datei angepasst. Der folgende Codeausschnitt zeigt die Implementierung des `transformation.result` Components.

```
<div class="container">
  <li *ngFor="let item of results; index as i" class="list-results">
    <app-transformation-result-item [text]="item"></app-transformation-
        result-item>
  </li>
</div>
```

In Abbildung 17 ist die Transformation der Query, sowie das Ergebnis abgebildet.

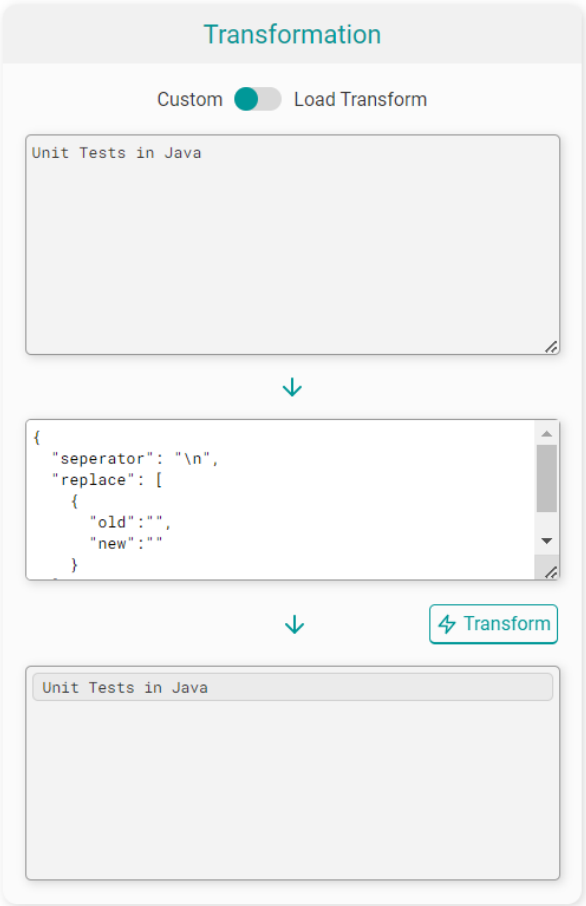

Abbildung 17: Transformation Kachel im Frontend

Der dritte Bereich der Website stellt die Auswahl und Ergebnisauflistung des KI-Services dar. Zur Datenakquirierung werden in Angular Services verwendet. Diese sind TypeScript Dateien, die Kommunikation zur API regeln. Eine der Funktionen aus dem Service für die Service-Component ist `getService()` Methode.

```
getServices() {
  const headers = {'Authorization': environment.token}
  return this.http.get<any>(environment.backend_url + '/service', {
     headers})
}
```

Diese kann innerhalb der TypeScript Datei der Component aufgerufen werden. Die `getService()` Methode stellt einen HTTP GET-Request an die `/service` Route der API. Die genaue URL der API wird aus den Environmentvariablen ausgelesen. Die Methode gibt jedoch nicht direkt die Response auf den Request zurück, sondern ein Objekt vom Typ Observable. Innerhalb des TypeScript Datei des Components kann auf dieses Observable subscribed werden. Durch diese Herangehensweise wird ein asynchroner Aufruf an die API gesendet und bei einer Response das Ergebnis verarbeitet. Das Ergebnis ist hierbei eine Liste aller verfügbaren KI-Services. Diese werden in der Variable `services` gespeichert.

```
this.serviceService.getServices().subscribe(res => {
  this.services = res['services']
})
```

Im HTML des Service-Components wird die Liste der Services für ein Dropdown verwendet. Eine weitere Funktion von Angular sind Events. Ein Event kann von einem Component ausgelöst werden. Innerhalb des Components, dass das untergeordnete Component implementiert, kann auf dieses Event reagiert werden. Es besteht die Möglichkeit TypeScript Code auszuführen oder vordefinierte Funktionen aufzurufen. Im Dropdown wird das `changeEvent` abgefangen. Dieses löst aus, wenn der Nutzer ein neues Element aus dem Dropdown ausgewählt hat. Dabei wird die Funktion `serviceSelected()` aufgerufen.

```
<app-dropdown [elements]="services" (changeEvent)="serviceSelected(
   $event)"></app-dropdown>
```

Die Tabelle für die Parameter nutzt ebenfalls `*ngFor`, um die einzelnen Reihen zu generieren.

Der letzte Component innerhalb des Services ist der `service.result` Component. Dieser zeigt ein Ergebnis aus dem KI-Service an. Das Ergebnis besteht dabei aus einer Überschrift und einem Body. Damit die Ergebnisse beim Frontend eintreffen, muss der Nutzer nach der Auswahl des Services auf den „Execute“ Button klicken. Dadurch wird im Angular Service die Methode `sendRequests()` aufgerufen, die dafür sorgt, dass die Anfrage für den KI-Service an die API gesendet wird. Anschließend wird die Funktion `pollResponses(expectedResponses: number)` mit der Anzahl der erwarteten Ergebnisse aufgerufen. Diese führt in einem 200ms Intervall die `poll()` Funktion aus, die eine Anfrage an die API stellt, ob bereits neue Ergebnisse des KI-Service im Backend zwi-

schengespeichert wurden. Falls dies der Fall sein sollte, werden die Ergebnisse in der Response mitgeteilt und die Anzahl der erhaltenen Ergebnisse hochgezählt. Die Schleife läuft so lange, bis die Anzahl der erhaltenen Ergebnisse gleich der Anzahl der erwarteten Ergebnisse ist. Für jedes Ergebnis wird ein `service.result` Component in einer `*ngFor` Schleife erzeugt. Die Auflistung der Ergebnisse, sowie die zuvor beschriebenen Elemente der „Service“ Kachel, sind in Abbildung 18 zu sehen.

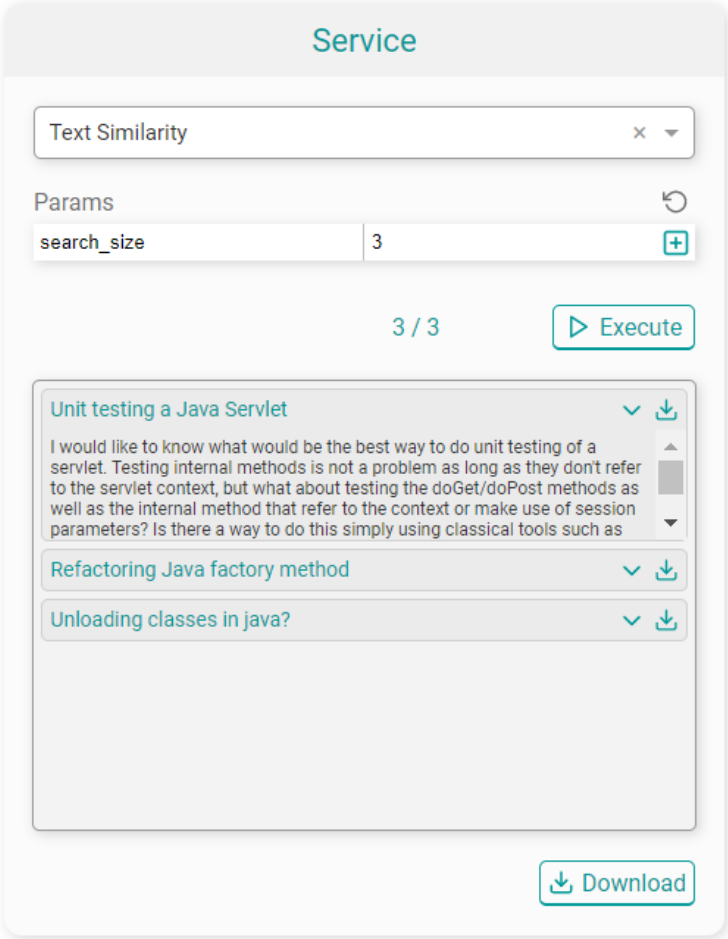

Abbildung 18: Service Kachel im Frontend

4.4 Deployment der Software mit Docker

Eine der Anforderungen der CONET war es, dass die entwickelte Schnittstelle Plattformunabhängig genutzt werden kann. Aus diesem Grund wurden alle Bereiche der Softwarearchitektur über Docker Container deploybar gemacht. Um eine eigens entwickelte Software in einen Container zu packen, muss zunächst ein Image aus der Software erstellt werden. Ein Image dient als Bauanleitung für den Container und enthält alle Dateien, die zur Ausführung des Programms benötigt werden. Ein Image kann über eine Dockerfile generiert werden. Die Dockerfile zur Erstellung des Images für das Backend ist nachfolgend abgebildet.

```
FROM python:3.10-slim
WORKDIR /app
```

```
ADD . /app
RUN pip install -r requirements.txt
EXPOSE 80
CMD ["python", "app.py"]
```

Innerhalb der Dockerfile können Pakete aus dem Dockerhub als Abhängigkeit angegeben werden, die beim bauen des Images heruntergeladen werden. Im Falle des Backendes wird das Paket `python:3.10-slim` bezogen. Innerhalb der Dockerfile ist ebenfalls die Festlegung der Ordnerstruktur, sowie die Ausführung von Kommandos möglich.

Zum Bauen eines Images mithilfe der zuvor erstellten Dockerfile wird der `build` Befehl benötigt. Für das Backend, den KI-Service, sowie das Frontend ist eine Docker-Compose Datei zuständig. Die Docker-Compose Datei kann in einer Linux- oder Windows-Subsystem für Linux (WSL) Umgebung ausgeführt werden. Der dazu nötige Befehlt lautet:

```
sudo docker compose -f '/path/to/compose/' build
```

Anschließend existieren drei Images in der Dockerumgebung, die zum Bau der Container genutzt werden können.

```
version: '3.3'
services:
    flask-backend:
        build:
            context: ../BA-Backend/
            network: host
    flask-services:
        build:
            context: ../BA-Services/
            network: host
    frontend:
        build:
            context: ../BA-Frontend/
            network: host
```

Das nachfolgende Auszug aus der `docker-compose.core.yml` Datei zeigt die Bauanleitung für einen Container der MySQL Datenbank. Innerhalb dieser Compose-Datei wird zunächst der Name des Services definiert. Unter diesem Namen ist der Docker Container im Netzwerk durch andere Docker Container erreichbar. Innerhalb des Services stehen mehrere Optionen zur Festlegung von Eigenschaften des Containers zur Verfügung. Über `ports` können ein oder mehrere Ports freigeschaltet werden, sodass der Container Zugriffe von außerhalb erlaubt. Im Bereich `volumes` stehen alle Ordner oder Dateien, die persistent gespeichert werden. Bei MySQL gibt es die Möglichkeit, ein initiales SQL-Skript anzugeben, welches nur bei der ersten Erstellung des Docker Containers ausgeführt wird. Das SQL-Skript erstellt eine Datenbank und die benötigten Tabellen. Im `image` Bereich ist das Image zum Bauen des Containers angegeben. Dies ist entweder ein lokal vorhandenes oder ein im Docker Hub angebotenes Image. Der letzte aufgeführte Punk `networks` beinhaltet das Netzwerk, in dem sich der Docker Container befindet. Dieses wird benö-

tigt, damit unterschiedliche Container miteinander kommunizieren können. Dazu muss sichergestellt werden, dass alle Container im gleichen Netzwerk sind.

```
services:
    mysql:
        ports:
            - '3333:3306'
        volumes:
            - mysql-conf:/etc/mysql/conf.d
            - mysql-storage:/var/lib/mysql
            - ./setup.sql:/docker-entrypoint-initdb.d/init.sql:r
        [...]
        image: mysql:latest
        networks:
            - custom-network
networks:
    custom-network:
        driver: bridge
        name: custom-network
volumes:
    mysql-storage:
    mysql-conf:
```

Der Container kann mit folgendem Befehlt gebaut und gestartet werden:

```
sudo docker compose -f '/path/to/compose/' up -d
```

Sollte der Container anschließend wieder beendet werden, so wird folgender Befehl benötigt:

```
sudo docker compose -f '/path/to/compose/' down
```

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.

DANIEL, R., 2018. *Schlagwort: Software-Anforderungen IEC 62304 konform dokumentieren* [online]. [besucht am 2022-12-31]. Abger. unter: `https://www.johner-institut.de/blog/tag/software-anforderungen/`.

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.

ENDRES, A., 2000. „Open Source“ und die Zukunft der Software. *Informatik-Spektrum*. Jg. 23, Nr. 5, S. 316–321.

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.

HAMET, P.; TREMBLAY, J., 2017. Artificial intelligence in medicine. *Metabolism*. Jg. 69, S36–S40.

HOQUE, N.; BHATTACHARYYA, D.; KALITA, J., 2015. Botnet in DDoS attacks: trends and challenges. *IEEE Communications Surveys & Tutorials*. Jg. 17, Nr. 4, S. 2242–2270.

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.

JOHANSSON, L.; DOSSOT, D., 2020. *RabbitMQ Essentials*. RabbitMQ Essentials: Build Distributed and Scalable Applications with Message Queuing Using RabbitMQ. Birmingham: Packt Publishing, Limited. ISBN 9781789131666.

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.

NEWMAN, S., 2015. *Microservices: Konzeption und design*. MITP-Verlags GmbH & Co. KG.

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.

SNEED, H.M., 2006. Integrating legacy software into a service oriented architecture. In: *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE, 11–pp.

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

WOLFF, E., 2018. *Microservices: Grundlagen flexibler Softwarearchitekturen*. dpunkt. verlag.