Vergleich von lexikalischer und semantischer Suche in einem Retrieval Augmented Generation (RAG) System

DIPLOMARBEIT

Höhere Abteilung für Informatik

01/10/2024 – 01/04/2025

Projektmitglieder: Sebastian Horner
Moritz Kern
Martin Pilgerstorfer
Tobias Wahl

Betreuer:in: Ing. Dominik Raffetseder, MSc

In Zusammenarbeit mit: ITPRO Consulting & Software GmbH
Delia Dorn, BA

Eidesstattliche Erklärung

Hiermit versichern wir, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von uns angegebenen Quellen angefertigt zu haben. Alle Stellen, die wörtlich oder sinngemäß aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Bei der Erstellung der Arbeit haben wir generative KI-Tools wie ChatGPT oder DeepL zu folgendem Zweck verwendet: Rechtschreib- und Grammatikprüfung, Generierung von Code-Kommentaren.

Gendererklärung

Um die Lesbarkeit dieser Diplomarbeit zu erleichtern, werden geschlechtsspezifische Bezeichnungen, die sowohl Frauen als auch Männer einschließen, in der üblichen männlichen Form verwendet. Dies geschieht ausschließlich aus Gründen der sprachlichen Vereinfachung und soll in keiner Weise eine Benachteiligung oder Missachtung des Gleichheitsgrundsatzes darstellen.

Danksagung

Wir möchten uns herzlich bei allen Personen bedanken, die uns während der Erstellung dieser Diplomarbeit unterstützt haben.

Besonders bedanken möchten wir uns bei unserm Betreuer Prof. Ing. Dominik Raffetseder, MSc, der uns während der Ausarbeitung der Diplomarbeit wertvolle Tipps gegeben hat.

Ein weiteres Dankeschön gilt unserem Auftraggeber, der ITPro. Besonders bedanken wir uns bei den technischen Ansprechpartnern David Aberl und Iris Schlager, die uns während der Entwicklung unserer Arbeit bei technischen Herausforderungen unterstützt haben. Sie haben uns bei Problemen unterstützt und uns wertvolle Lösungsansätze vermittelt und uns geholfen das Projekt erfolgreich umzusetzen.

Ebenso möchten wir uns bei Delia Dorn bedanken, die die Organisation und Leitung der regelmäßigen Meetings übernommen hat, die besprochen Inhalte immer mitgeschrieben hat und anschließend für uns bereitgestellt hat.

Impressum

Schule
HTBLA Perg für Informatik
Machlandstraße 48
4320 Perg

Schuljahr
2024/25

Klasse
5BHIF

Projekttitel
Vergleich von lexikalischer und semantischer Suche in einem Retrieval Augmented Generation (RAG) System

Projektteam:
Sebastian Horner
Moritz Kern
Martin Pilgerstorfer
Tobias Wahl

Betreuungslehrer
Prof. Ing. Dominik Raffetseder, MSc

Auftraggeber
ITPRO - Consulting & Software GmbH

Abstract

Nowadays, searching for information in documents is particularly challenging due to the enormous amount of stored files. Our thesis presents a company-specific Retrieval-Augmented Generation (RAG) system that replaces the conventional full-text search. It allows users to ask questions to which the system searches for answers in a large collection of files.

The system contains two databases as sources of information: Elasticsearch for a lexical search, and Milvus for a vector search. These are filled with identical data for comparison and analysed in the course of this thesis.

Files that are to be provided for the search are stored in a folder and automatically extracted, transformed and inserted into the databases by a service. This folder is also monitored and changes are automatically transferred.

If a user asks a question in the front end, significant word groups are determined with the help of artificial intelligence. After further processing, the search is carried out and the system receives a section of a file from the information sources. This section is sent back to the artificial intelligence and the user question is answered in natural language.

A web interface has been developed as the GUI. Only authorised people have access. Users have their own chats and can ask questions. It is also possible to expand the context of the search in the background with your own text.

The individual system components are hosted in Docker containers. Centralised control takes place via the backend, which provides a REST API and processes all HTTP requests.

Zusammenfassung

Die Suche nach Informationen in Dokumenten gestaltet sich heutzutage, aufgrund der enormen Menge abgelegter Dateien, als besonders herausfordernd. Unsere Diplomarbeit präsentiert ein unternehmensspezifisches Retrieval-Augmented Generation (RAG)-System, das die herkömmliche Volltextsuche ablöst. Damit können Benutzer Fragen stellen, auf die das System in einer großen Sammlung von Dateien gezielt nach Antworten sucht.

Das System beinhaltet zwei Suchdatenbanken als Informationsquelle: Elasticsearch für eine lexikalische Suche, und Milvus für eine Vektorsuche. Diese werden zum Vergleich mit identischen Daten befüllt und im Laufe dieser Diplomarbeit analysiert.

Dateien, die zur Suche bereitgestellt werden sollen, werden in einem Ordner abgelegt und von einem Service automatisch extrahiert, transformiert und in die Datenbanken eingefügt. Dieser Ordner wird auch überwacht und Änderungen automatisch übertragen.

Stellt ein Benutzer im Frontend eine Frage werden mithilfe künstlicher Intelligenz signifikante Wortgruppen ermittelt. Nach einer Weiterverarbeitung wird eine Suche durchgeführt und das System bekommt aus den Informationsquellen einen Abschnitt einer Datei. Dieser Abschnitt wird wieder an die künstliche Intelligenz gesendet und die Benutzerfrage in natürlicher Sprache beantwortet.

Als GUI ist eine Weboberfläche entwickelt worden. Zugriff haben nur autorisierte Personen. Benutzer erhalten eigene Chats und können Fragen stellen. Zusätzlich besteht die Möglichkeit, den Kontext der Suche im Hintergrund mit einem eigenen Text zu erweitern.

Die einzelnen Systemkomponenten werden in Docker-Containern gehostet. Die zentrale Steuerung erfolgt über das Backend, das eine REST-API bereitstellt und alle HTTP-Requests verarbeitet.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Das Unternehmen ITPRO stellt Schülern jährlich einen Ideenkatalog für Diplom- und Projektarbeiten zur Verfügung. In der vierten Klasse hat sich unser Team für die Idee einer lexikalischen Suche in einem RAG-System entschieden, mit der Intention, dieses Projekt als Diplomarbeitsprojekt weiterzuführen.

Die Idee besteht darin, unsortierte Dateien, gespeichert in einem Ordner, effizient und schnell durchsuchen zu können. Das Unternehmen arbeitet im Consulting- und Entwicklungsbereich und hat dementsprechend viele Kunden. Das entwickelte Projekt nutzt die firmeneigenen Sicherheitsmechanismen, sodass alle autorisierten Benutzer das RAG-System verwenden können.

Ein vorangegangenes Projekt in der vierten Klasse diente als Prototyp und findet keinerlei Verwendung mehr im Diplomarbeitsprojekt. Die finale Anwendung wurde mit den bereits gewonnenen Erfahrungswerten neu aufgebaut.

1.2 Zielsetzung

Ziel des Projekts ist es in vielen Dateien gleichzeitig nach einer genauen Antwort zu suchen, auch wenn dem Benutzer nicht genau klar ist, in welchem Dokument sich die Informationen zur Beantwortung der Frage befinden. Dieser Abfrageprozess wird mit zwei Datenbanken realisiert und zum Vergleich gegenübergestellt und analysiert.

Bisher ist die herkömmliche Suche in Dokumenten auf die Volltextsuche beschränkt. Diese ist meistens auf ein Dokument angewandt worden und es war nur die Suche nach einem bestimmten Wort bzw. einer Wortgruppe möglich.

Mit dem Retrieval-Augmented Generation-System soll es möglich sein, dass man den Aufbau und Ort der gesuchten Informationen nicht genau kennt und trotzdem die genaue Antwort auf eine Frage erhält. Durchsuchte Dokumente sollen in einem Ordner liegen, welcher dauernd auf Änderungen überwacht wird.

Das finale Produkt, welches als Ergebnis der Diplomarbeit gilt, ist eine Fullstack Anwendung. Ein Angular Frontend ist für die Anzeige der Chats zuständig. Die Kommunikation zwischen Schnittstellen, die Businesslogik und das Einfügen der Dokumente in die Suchdatenbanken übernehmen zwei .NET-Anwendungen. Zu Analyse und Vergleichszwecken werden die Datenbanken Elasticsearch und Milvus verwendet.

1.3 Projektinhalt - Überblick

Das Projekt ist in vier Teile aufgeteilt. Die Anwendung besteht aus einem Backend, Frontend und zwei Suchdatenbanken. Das gemeinsame Zusammenspiel ist essenziell für die Beantwortung einer Benutzerfrage.

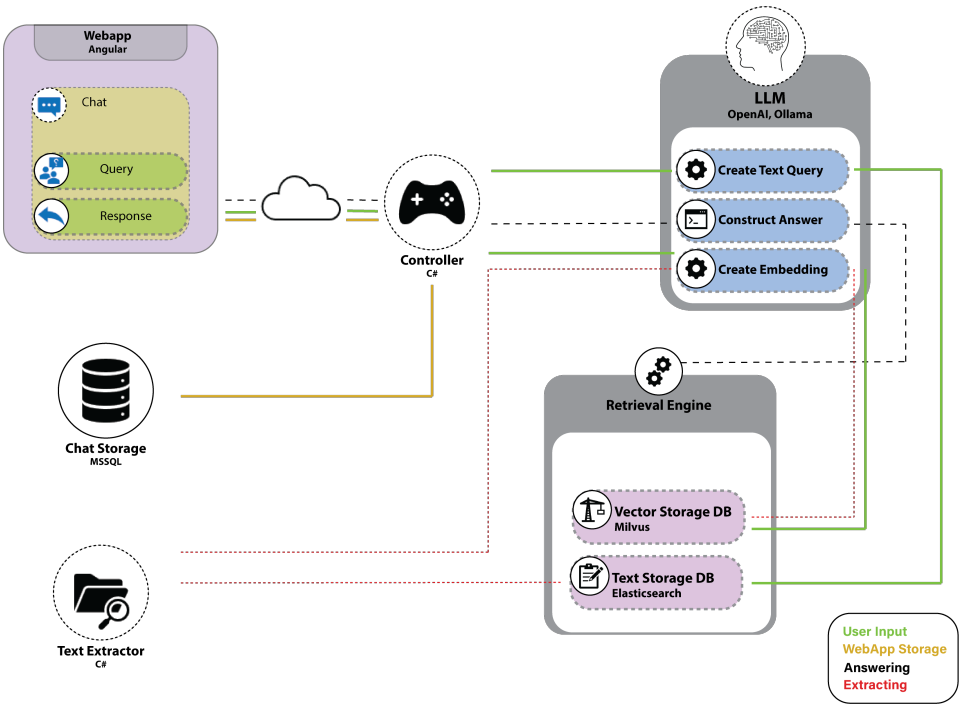


Abbildung 1: Aufteilung und Kommunikation des RAG-Systems

1.3.1 Datenbanken

Die Datenbanken **Milvus und Elasticsearch** bilden das Herzstück des Systems. Sie enthalten die essenziellen Informationen, um Benutzerfragen zu beantworten. Die Datenbanken selbst werden von außen über API-Schnittstellen aufgerufen, dabei wird jede Abfrage nur an eine Datenbank gesendet. Intern wird eine Suche durchgeführt und Abschnitte mit einer großen **Trefferquote** zurückgegeben.

1.3.2 Frontend

Die Schnittstelle zum Benutzer bildet das **Frontend**. Diese GUI ist im **Chat-Design** aufgebaut. Jeder Benutzer kann sich über das firmeninterne SSO an der Anwendung anmelden und seine Chats und Fragen werden im Hintergrund in einem **Chat Storage** gespeichert. Dem Benutzer ist es möglich, neue Chats anzulegen und zu löschen und darin Fragen zu stellen.

Ebenfalls besteht die Option, die Suche bei jeder Frage noch mit einem Content zu erweitern. Für jede Benutzerfrage werden die wichtigsten Wörter von einer künstlichen Intelligenz im Hintergrund markiert und vervielfältigt. Die Art, diese Wörter zu markieren und vervielfältigen, kann mit dem Content bearbeitet werden.

1.3.3 Backend

Das **Backend** besteht aus zwei Anwendungen. Der **Controller** übernimmt die Netzwerkkommunikation sowie die Geschäftslogikprozesse. Der **DocumentDissector** (Abb. 1 „TextExtractor“) übernimmt die Extraktions- und Transformationsprozesse.

Der Controller stellt im ChatService die Verwaltung der Chats im Frontend zur Verfügung. Für die Beantwortung ist der AnsweringService zuständig. Der Zugriff auf beide Services ist nur für autorisierte Benutzer möglich. Nach außen hin stellt der Controller eine REST-API zur Verfügung, damit das Frontend die Services nutzen kann.

Die Beantwortung wird mithilfe von **künstlicher Intelligenz** durchgeführt. OpenAI erkennt signifikante Wörter einer Benutzerfrage und ist für die Beantwortung in natürlicher Sprache zuständig.

Der **DocumentDissector** verwaltet einen Ordner, der als Informationsquelle dient. Dort gespeicherte Dokumente werden extrahiert, transformiert und in die Suchdatenbanken integriert. Änderungen an den Originaldaten werden automatisch durch dieselben Prozesse berücksichtigt.

1.4 Projektumfeld

1.4.1 Projektteam

Abbildung 2: Projektteam

Das Projektteam stellt sich aus Sebastian Horner und Moritz Kern aus Schönau im Mühlkreis, Martin Pilgerstorfer aus Mauthausen und Tobias Wahl aus Naarn im Machlande zusammen. Die genaue Aufteilung der Aufgaben findet sich weiter unten.

Abbildung 2: v. l. n. r.: Sebastian Horner, Moritz Kern, Tobias Wahl und Martin Pilgerstorfer.

Aufgabenverteilung siehe hier

1.4.2 Auftraggeber

Auftraggeber dieser Diplomarbeit ist das Softwareentwicklungsunternehmen ITPRO aus Linz. Neben dem Hauptstandort in Linz gibt es zwei weitere Standorte, Hagenberg und Ottensheim. Auftraggeberseitig unterstützten uns hauptsächlich Delia Dorn, BA. im organisatorischen Bereich sowie David Aberl, MSc. im technischen Bereich sowie bei der genauen Ideenausarbeitung bzw. Umsetzungsplanung. Im Zuge eines Ferialpraktikums im Sommer 2024 konnte das Projektteam den Implementierungsteil der Diplomarbeit direkt beim Auftraggeber vor Ort in Linz fertigstellen. Dabei wurde das Projektteam sehr gut in das Team „Gravity“ eingebunden und es konnte bei technischen sowie allgemeinen Fragen immer Rücksprache gehalten werden.

IT | PRO
Software die's einfach macht.

Abbildung 3: ITPro Logo

1.4.3 Betreuung

Die Diplomarbeit wurde im technischen und theoretischen Teil von Ing. Dominik Raffetseder, MSc. betreut. Das Projektteam stand während der Planung, der Implementierungsphase sowie beim Schreiben der Diplomarbeit in ständigem Austausch mit ihm. Durch sein tiefgründiges Wissen im Bereich der Datenbanken sowie verschiedenster Webtechnologien war er sehr gut geeignet, diese Diplomarbeit zu betreuen. Wir möchten ihm großen Dank für seine Unterstützung aussprechen.

1.5 Ausgangslage

In der 4.Klasse ist das Projekt „Panda - Project for advanced natural document answering“ gestartet. Die Idee, eine schnelle Dokumentenabfrage mit künstlicher Intelligenz zu bauen, stammt vom Unternehmen ITPRO.

In der Anfangsphase der Entwicklung bis zur Diplomarbeit ist ein Prototyp entstanden. Dieser hat die Grundfunktionen bereits abgebildet, ist danach aber noch um einiges erweitert und umstrukturiert worden.

Das Frontend ist aus einem Chat bestanden, in welchem man Fragen stellen konnte. Diese Fragen sind nicht persistiert und keinem Benutzer zugeteilt worden. Ebenfalls ist das SSO, der Sicherheitsmechanismus, der Firma noch nicht eingebaut worden.

Die Dokumente sind zum Beginn der Diplomarbeit nur in Elasticsearch gespeichert worden. Die Indizierung hat eine ausgelagerte Software übernommen, das Format der gespeicherten Informationen ist von dem Tool vorgegeben worden. Die Milvus Datenbank ist im Vorprojekt noch nicht verwendet worden.

Das Backend hat die Beantwortung einer Benutzerfrage bereits durchführen können, dabei ist die Kommunikation zum Frontend nur über einen REST-Endpunkt gelaufen. Hat ein Benutzer eine Frage gesendet, ist diese noch nicht von OpenAI transformiert worden. Das LLM hat lediglich einen Query-String für Elasticsearch gebaut und die Frage eins zu eins eingefügt. Dabei sind die Felder des Elasticsearch-Index mitgesendet worden, um einen korrekten Query bauen zu können. Die Beantwortung der ursprünglichen Frage mithilfe von Informationen eines Dokuments hat funktioniert, ist aber optimiert worden.

Die Chat-Datenbank ist erst im Laufe der Diplomarbeit erstellt worden. Dazu gehört auch die gesamte Verwaltung im Backend, vom DB Kontext über die Migrationen bis hin zu der Verwendung mit EF Core. Um die Datenbankoperationen vom Frontend aus aufrufen zu können, sind einige REST-Endpunkte hinzugefügt worden. Dafür wurde ein neues Service erstellt und einige REST-Endpunkte sind hinzugefügt worden. Das Backend ist in der Projektstruktur komplett verändert worden, Datenmodelle wurden ausgelagert und die Funktionen in die Projekte Businesslogik und Controller aufgeteilt. Auch die Authentifizierung und Autorisierung mittels SSO ist erst im Diplomarbeitsprojekt hinzugekommen.

2 Theoretische und fachpraktische Grundlagen und Methoden

2.1 Grundlegende Fachbegriffe

2.1.1 Retrieval-Augmented Generation (RAG) System

RAG-Systeme erweitern die leistungs- und ressourcenstarken Large Language Models, um Textgenerierung in Bezug auf benutzerdefinierte Daten in Unternehmensbereichen voranzutreiben.

LLMs sind eine Art von KI-Modellen, die mithilfe von immens großen Datensätzen natürliche Sprache erkennen und Antworten generieren. Diese Programme stellen Prozesse des maschinellen Lernens dar, die auf neuronalen Netzen basieren und Deep-Learning-Techniken verwenden. Deren Entwicklung wird federführend vom Unternehmen OpenAI vorangetrieben.

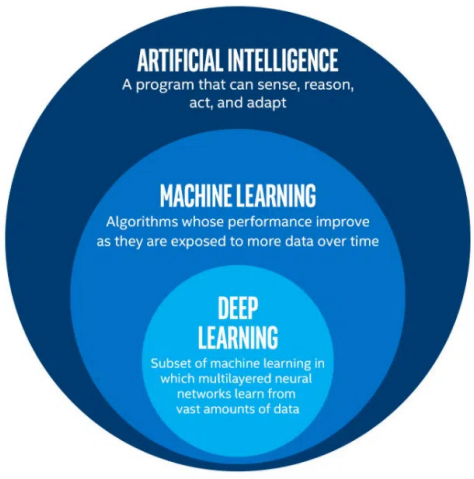


Abbildung 4: Einteilung Funktionen des Deep Learnings in Künstliche Intelligenz [1]

LLMs bieten jedoch den Nachteil, zeitweise zu halluzinieren. Das heißt, dass sie Antworten generieren, die grammatikalisch korrekt, faktisch jedoch falsch sind. Dieses Phänomen hat mehrere erklärbare Gründe (vgl. [2]):

- **Unvollständige oder rauschende Daten:** Die Quelle von gelernten Daten ist nicht immer bekannt und kann auch nicht auf faktische Korrektheit überprüft werden. Sie können inkorrekt, nicht aktualisiert oder unvollständig sein.
- **Ungenaue Fragestellung:** Die Fragestellung kann im Kontext unklar formuliert sein. Ein LLM kann darauf schlecht reagieren.
- **Overfitting oder Underfitting:** Zwei Phänomene im Bereich des maschinellen Lernens, welche das zu genaue oder zu generalisierte Lernen von Daten beschreibt.
- **Inhärente Vorurteile (Bias):** Modelle können Vorurteile aus Trainingsdaten übernehmen und faktisch inkorrekte Annahmen treffen.
- **Semantische Lücken:** Wir Menschen verfügen über einen gesunden Menschenverstand und arbeiten diesen in Dokumenten ein, um nicht alles genauestens erklären zu müssen. LLMs arbeiten mithilfe von Pattern-Erkennung und haben die Fähigkeit nicht, diese Wissenslücken zu erkennen. Auch dadurch kann es zu unwahren Annahmen kommen.

RAG-Systeme, erstmals beschrieben 2020 im Paper „Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks [3]“, gehen gegen diese Halluzinationen vor. Im Prozess der Retrieval-Augmented Generation werden externe Wissensressourcen verwendet, um die Modelle in diesen Themenbereichen zu vervollständigen.

Wo dieses externe Wissen abgerufen und in den Prozess eingearbeitet wird, obliegt dem Entwickler. Es gibt dabei verschiedene Konzepte, wobei unser Projekt Prompt-Engineering-Techniken benutzt. (siehe Abbildung 24).

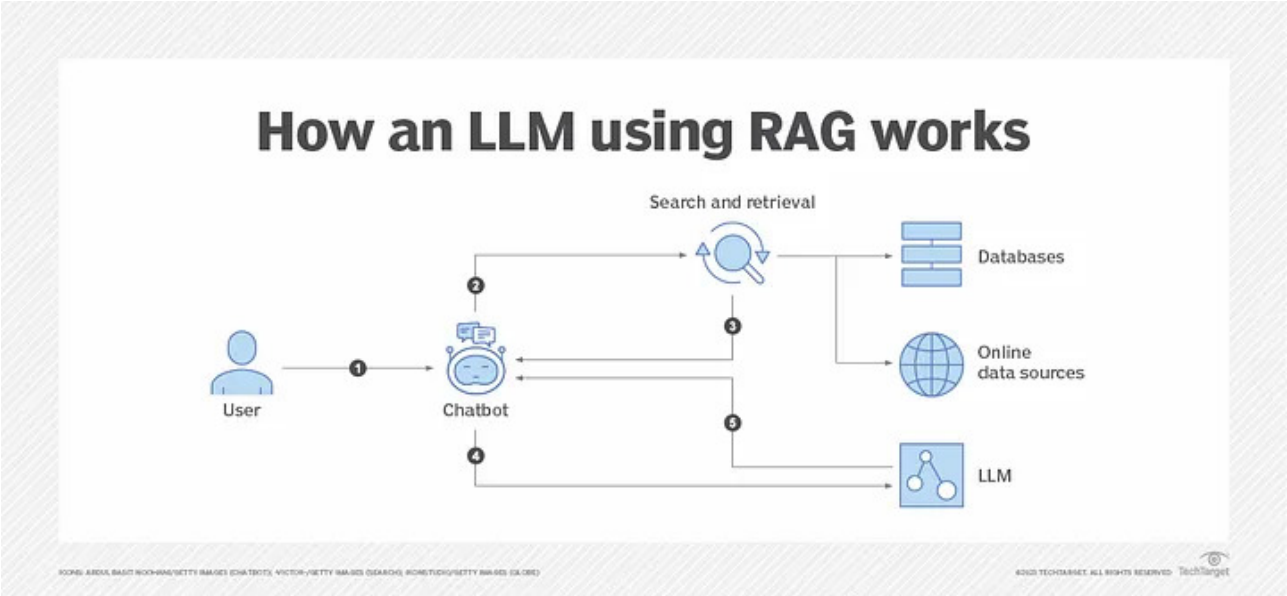


Abbildung 5: RAG-System Beispielablauf [4]

2.1.2 Lexikalisch

Lexikalisch bedeutet grundsätzlich, ohne Zusammenhänge der einzelnen Wörter in einem Lexikon zu suchen. Im Falle von Elasticsearch ist der invertierte Index als Lexikon zu verstehen, in dem bei der Suche nach einzelnen Wörtern gesucht wird, um herauszufinden, wo und in welchen Dokumenten die gesuchten Wörter vorkommen. Mehr Infos im Kapitel Elasticsearch.

[5] [6]

2.1.3 Semantisch

Semantisch bezieht sich auf die Bedeutung und die Zusammenhänge von Wörtern oder Phrasen, anstatt nur auf deren exakte Schreibweise. Mithilfe von Milvus, einer Vektordatenbank, kann man nach Wörtern suchen, die nicht exakt vorkommen. Das bedeutet, dass Begriffe mit ähnlicher Bedeutung erkannt werden, selbst wenn sie nicht genau so im gesuchten Text vorkommen. [7]

2.2 Verwendete Technologien

2.2.1 Python

Für kleinere Programmieraufgaben, etwa ein Hilfsprogramm für Datenbankanalysen (siehe Analyzer), kam die weit verbreitete Sprache Python zum Einsatz. Python wurde ursprünglich von Guido van Rossum entwickelt und ist 1991 erschienen. Dank seiner Einfachheit und der umfangreichen Community-Ressourcen ist die Lernkurve flach – weshalb es häufig für kleine Projekte genutzt wird, aber auch für größere Aufgaben geeignet ist, da objektorientierte Programmierung unterstützt wird.

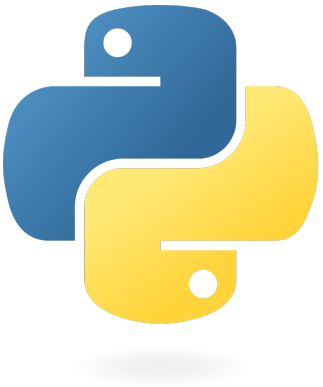


Abbildung 6: Python Logo [8]

Website: Python.org

[8] [9]

2.2.2 Angular

Angular ist ein TypeScript-basiertes Open-Source-Web-Framework zur Entwicklung von Single-Page Applications (SPAs). Es basiert auf einer komponentenbasierten Architektur und bietet eine umfangreiche Auswahl an Tools, APIs und Bibliotheken, die den Entwicklungsprozess erleichtern. Angular wird von Google und der Community entwickelt und betreut. [10]

Abbildung 7: Angular Logo

2.2.3 Docker

Docker ist vor allem unter Softwareentwicklern ein bekanntes und beliebtes, plattformunabhängiges Programm zur Virtualisierung von Anwendungen auf Betriebssystemebene. Einzelne ausgeführte Programme laufen in sogenannten Docker-Containern, die nahezu vollständig isoliert sind. Docker-Compose-Dateien ermöglichen es, mehrere Container gemeinsam zu starten.

Abbildung 8: Docker Logo [11]

Ein großer Vorteil von Docker ist, dass eine Anwendung unabhängig vom Betriebssystem stets lauffähig ist und sich immer gleich verhält, da die Container auf dem gleichen Kernel laufen - in der Regel ein Linux-Kernel. Wird Docker unter Windows verwendet, muss daher Linux virtualisiert werden. Üblich ist hierfür die Verwendung des Windows Subsystem für Linux siehe WSL.

Website: Docker.com

[12] [11]

WSL

WSL ist ein Tool, dass es seit 2016 für Windows 10 und deren Nachfolger gibt. Das WSL ermöglicht es Linux Funktionen und Programme unter Windows auszuführen. Dabei wird zwischen der ersten Version (WSL1) und dem Nachfolger, der aktuellen WSL2 unterschieden. Die Idee bei WSL1 war, durch die sogenannte Kompatibilitätsschicht, Linux Befehle für Windows zu übersetzen, was zur Folge hatte, dass nicht alle Linux Funktionen und Programme verwendet werden konnten.

In der zweiten Version (WSL2) wird nun nicht mehr auf diese Kompatibilitätsschicht gesetzt, sondern mithilfe von Hyper-V ein gesamtes Linux-Betriebssystem virtualisiert. Da ein optimierter Linux-Kernel verwendet wird, können nun auch alle Systemfunktionen von Linux verwendet werden.

[14] [15] [13]

Abbildung 9: WSL Logo [13]

2.2.4 ASP.NET Core

ASP.NET Core ist ein modulares Open-Source Web-Framework von Microsoft. Es stellt mehrere Tools zur Verfügung, um alle möglichen Komponenten einer vollfunktionsfähigen Webapp erstellen zu können.

In unserem System wird im Backend die Web-API für die Erstellung von Controllern sowie das Entity Framework Core verwendet. Für die modulare Verwendung von Code sorgt Dependency Injection, welche je nach Anfrage bestimmte Entitäten mit anderen Klassenbeschreibungen deklariert. [16]

2.3 Verwendete Datenbanken

2.3.1 Elasticsearch

Was ist Elasticsearch?

Elasticsearch ist eine plattformunabhängige Open-Source Suchmaschine und NoSQL-Datenbank, die für die Speicherung und effiziente Abfrage von Dokumenten entwickelt wurde. Elasticsearch kann für Suchen, Analysen und Optimierungen sowie für Log-Management verwendet werden.

Kibana Kibana ist die Analyseplattform zu Elasticsearch. Die zum Elastic-Stack gehörende Webanwendung wird dazu verwendet, Daten zu visualisieren bzw. Elasticsearch bis zu einem gewissen Grad zu konfigurieren und zu managen.

Abbildung 10: Elasticsearch Logo

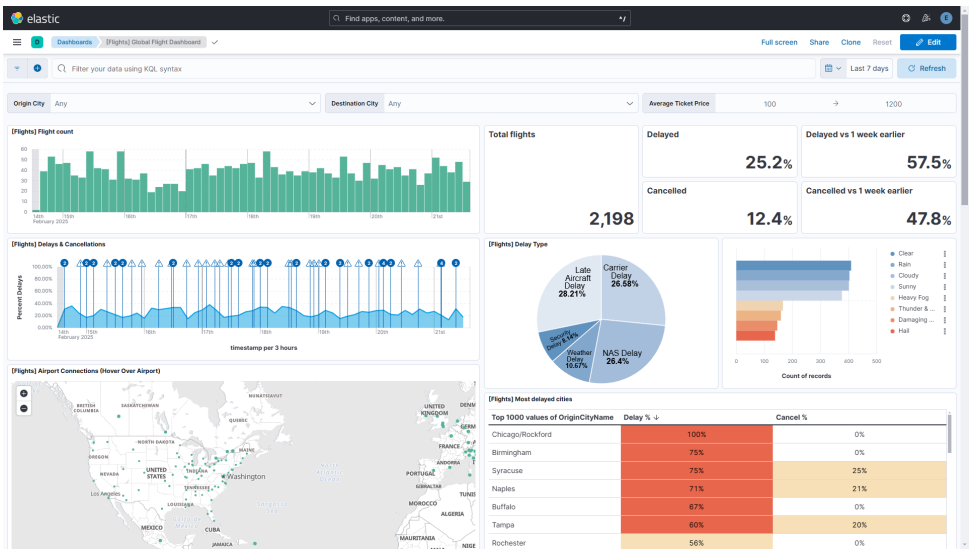


Abbildung 11: Kibana Beispieldashboard

Kibana bietet darüber hinaus auch Beispieldatenbanken an, die heruntergeladen werden können, damit Elasticsearch sowie Kibana ordentlich getestet werden können.

Das Kibana Flights Dataset beinhaltet zum Beispiel über 2000 Beispieldatensätze, sowie ein schönes Beispieldashboard, dass die Funktion von Kibana als Analyseplattform perfekt zur Schau stellt.

Logische Architektur

- **Cluster** Ein Cluster ist eine Zusammenfassung von einzelnen Knoten. Durch die Erhöhung der Knoten in einem Cluster steigt die Ausfallsicherheit, Skalierbarkeit und die Last wird besser verteilt. Jeder Cluster hat außerdem einen eindeutigen Namen, um identifiziert werden zu können. Der Cluster ist dann sozusagen das gesamte DBMS.
- **Knoten/Nodes** Ein Knoten ist eine einzelne Instanz von Elasticsearch in einem Cluster, welche Daten speichert und Suchanfragen ausführt.

Arten von Nodes

  - **Master-eligible Node** Diese Knoten können als Master arbeiten, es kann aber nur einen Master in einem Cluster geben. Der Master ist für die Cluster-Verwaltung zuständig. Die Cluster-Verwaltung beinhaltet Aufgaben wie die Aufgabenverteilung, Ressourcenüberwachung und Failover-Mechanismen.

  - **Data Node** Speichert Daten und führt Abfragen aus.

    Außerdem gibt es noch weitere Arten von Nodes, auf die nicht genauer eingegangen wird:

  - **Ingest Node**
  - **Remote-eligible node**
  - **Machine learning node**
  - **Transform node**

- **Indizes** Ein Index ist eine Sammlung von Dokumenten, die in Dokumenttypen unterteilt sind. Dokumente enthalten dann Felder, in denen die später durchsuchbaren Daten gespeichert werden.
  Verglichen mit anderen DBMS‘ wäre ein Index dort eine Datenbank. Wobei ein Typ dann eine Tabelle und ein Dokument eine Zeile wäre.
  Ab Version 7.x gibt es das Konzept von Dokumenttypen nicht mehr und es kann pro Index auch nur mehr ein Dokument-Typ gespeichert werden. In der Praxis hat sich hier „_doc“ als Standard durchgesetzt.
- **Shards** Ein Shard ist eine Teilmenge von Daten eines Index. Einzelne Shards können auf verschiedenen Knoten gespeichert werden, um die Last auf mehrere Knoten zu verteilen und die hohe Skalierbarkeit und Ausfallsicherheit zu realisieren.
- **Replicas** Ein Replikat ist eine Kopie eines Shards, welche auf einem anderen Knoten gespeichert wird. Dies erhöht die Ausfallsicherheit und ermöglicht parallele Verarbeitung von Suchanfragen.

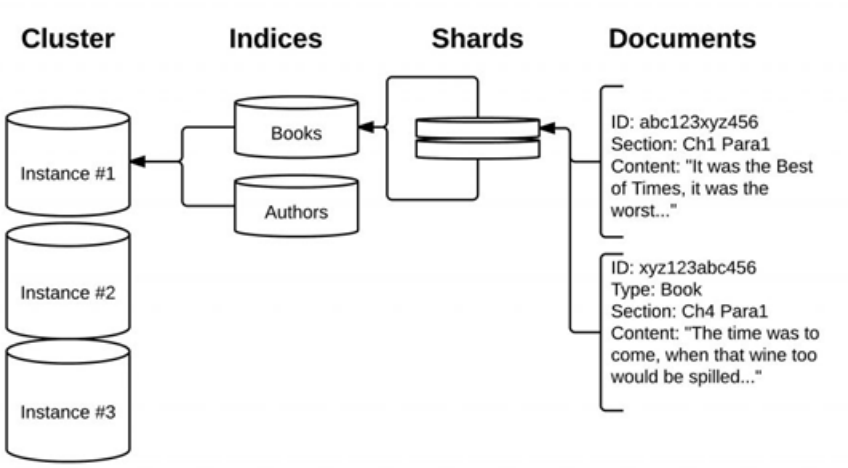


Abbildung 12: Logische Architektur

Physische Architektur

Die Nodes eines Clusters laufen auf verschiedenen Maschinen. Auf einer Maschine können theoretisch auch mehrere Nodes laufen. Innerhalb dieser Nodes gibt es dann die verschiedenen Shards der einzelnen Index sowie deren Replicas.

Abbildung 13: Physische Architektur

REST API

Die REST API ist die Schnittstelle zu Elasticsearch. Mit ihr können alle möglichen Funktionen verwendet und Abfragen erstellt werden. [17]

Die wichtigsten Funktionen der Elasticsearch REST-API sind:

1. **CRUD-Operationen:** Mit der Elasticsearch REST-API können alle CRUD-Operationen wie GET, POST, PUT und DELETE wie bei jeder anderen Datenbank durchgeführt werden.

2. **Aggregationen und Analysen:** Die Elasticsearch API erlaubt es außerdem direkt in den Abfragen Aggregationen und Analysen durchzuführen. Die Bucket Aggregationen erlauben es, Dokumente anhand verschiedener Kriterien in sogenannte „Buckets“ zu gruppieren.
3. **Verwaltung:** Die REST-API bietet umfassende Verwaltungsfunktionen, darunter:
    - **Cluster-Management:** Überwachung und Steuerung des Clusters, einschließlich Knotenzustand, Cluster-Health und Shard-Zuweisungen.
    - **Index-Management:** Erstellung, Löschung und Konfiguration von Indizes sowie Verwaltung von Mappings und Einstellungen.
    - **Alias-Management:** Verwaltung von Aliase für Indizes zur Vereinfachung von Abfragen und Operationen.
    - **Template-Management:** Definition von Vorlagen für Indizes, um konsistente Einstellungen und Mappings sicherzustellen.
    - **Snapshot- und Restore-Funktionen:** Erstellung und Wiederherstellung von Backups des Clusters oder einzelner Indizes.

Zu beachten ist, dass bei Requests immer die Authentifizierung berücksichtigt werden muss, daher ist Postman zum Testen sehr zu empfehlen. Außerdem können beim Testen mit Postman Variablen verwendet werden, was das Testen wesentlich einfacher und effizienter macht.

Umgekehrte Indexstruktur

Elasticsearch verwendet eine umgekehrte Indexstruktur, um effiziente Volltextsuchen zu ermöglichen. Der Prozess umfasst folgende Schritte:

- **Tokenisierung:** Der Text wird in einzelne Bestandteile (Tokens) zerlegt, beispielsweise in Wörter oder Phrasen.
- **Umformungen:** Die Tokens werden normalisiert, indem sie beispielsweise in Kleinschreibung umgewandelt oder um Sonderzeichen bereinigt werden.
- **Indexierung:** Die normalisierten Tokens werden in einer Datenstruktur gespeichert, die jedem Token die Positionen in den Dokumenten zuordnet, in denen es vorkommt. Dies ermöglicht schnelle Suchanfragen, da nicht jedes Dokument vollständig durchsucht werden muss.

[18] [19]

2.3.2 Milvus

Was ist Milvus?

Milvus ist eine Open-Source-Datenbank, die speziell für die Speicherung und Suche von Vektoren entwickelt wurde und auch als Cloud-Service verwendet werden kann. Sie basiert auf bekannten Suchbibliotheken wie Faiss, HNSW, DiskANN und SCANN und ermöglicht das effiziente Durchsuchen von Daten mit Millionen, Milliarden oder sogar Billionen von Vektoren. Milvus speichert und verwaltet Daten mithilfe von Vektoren und unterstützt verschiedene Datentypen wie Zahlen, Texte, Arrays und JSON, die zusätzlich zu den Vektoren gespeichert werden können. [20]

Abbildung 14: Milvus Logo

Vektoren und unstrukturierte Daten

Unstrukturierte Daten enthalten oft viele Informationen, die schwierig zu analysieren sind. Vektoren fassen die wichtigsten Merkmale solcher Daten kompakt zusammen. Milvus selbst erzeugt keine Vektoren, sondern speichert diese, nachdem sie von einem externen Modell, wie in unserem Fall mit OpenAI Text Embedding, in numerische Vektoren umgewandelt wurden.

Effiziente Speicherung und Suche

Die Datenbank ist darauf optimiert, Vektoren effizient zu speichern und in großen Datenmengen nach ähnlichen Vektoren zu suchen. Durch Optimierungen für verschiedene Hardwarearchitekturen ist Milvus besonders schnell. Die Suchmaschine wurde in C++ entwickelt, um eine hohe Leistung sicherzustellen.

Milvus-Architektur

Damit Milvus so schnell und effizient arbeiten kann, nutzt es eine Shared-Storage-Architektur. Das bedeutet, dass Speicher und Rechenleistung getrennt sind. Dadurch kann das System problemlos erweitert werden und bleibt auch dann stabil, wenn es viele Daten gleichzeitig verarbeiten muss. [21]

Die Architektur besteht aus vier Teilen:

1. Zugriffsschicht
2. Koordinatordienst

3. Arbeitsknoten
4. Speicher

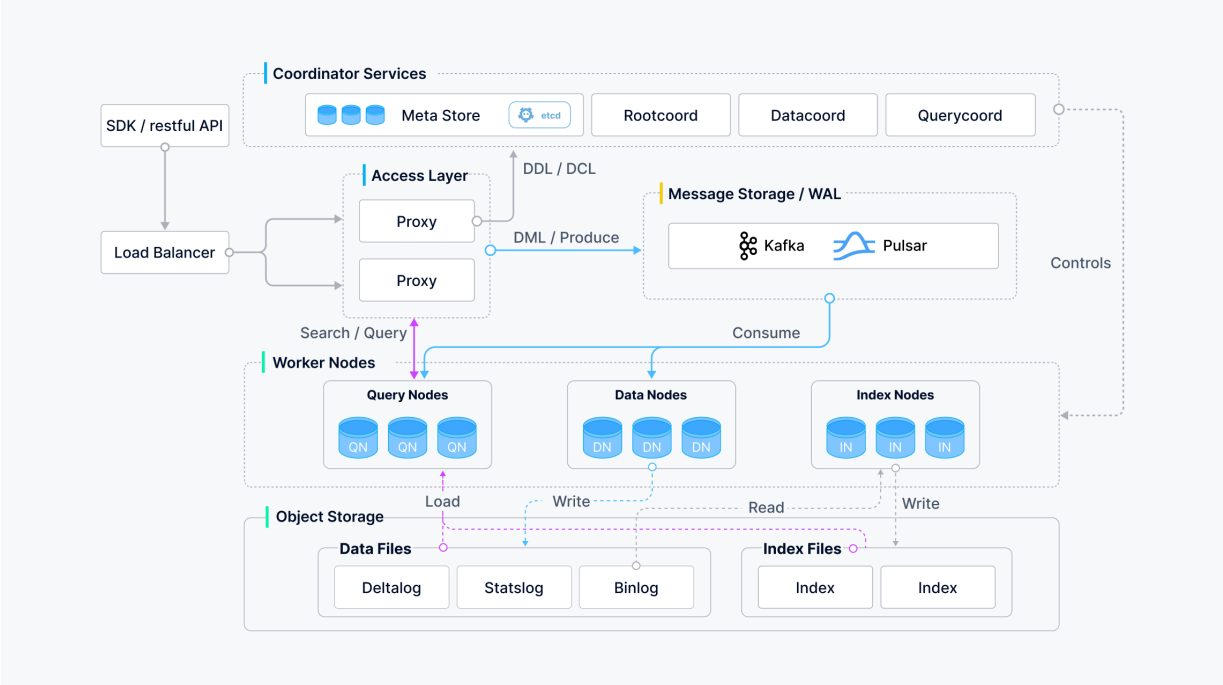


Abbildung 15: Milvus Architektur

Oben befindet sich die **Zugriffsschicht**, die Anfragen entgegennimmt. Darunter liegt der **Koordinatordienst**, der die Aufgaben steuert und verteilt. In der Mitte arbeiten die **Worker Nodes**, die für die eigentliche Berechnung verantwortlich sind. Ganz unten befindet sich der **Speicher**, der sowohl Metadaten als auch Vektordaten langfristig speichert.

Sicherheitsfunktionen und Datenschutz

Milvus unterstützt verschiedene Sicherheitsmaßnahmen:

- Rollenbasierte Zugriffskontrolle (RBAC): Einschränkung des Zugriffs auf Daten je nach Benutzerrechten.
- Datenverschlüsselung: Speicherung der Vektoren mit Verschlüsselung für erhöhte Datensicherheit.
- Audit-Logs: Nachvollziehbare Änderungen und Abfragen zur besseren Überwachung.

In vielen Anwendungsfällen ist es wichtig, dass Milvus DSGVO- oder HIPAA-konform eingesetzt wird, um personenbezogene Daten zu schützen.

Spaltenorientierte Datenbank

Milvus ist eine **spaltenorientierte Datenbank**, was bedeutet, dass Daten nicht zeilenweise, sondern spaltenweise gespeichert werden. Dies bringt erhebliche Vorteile für Suchanfragen und Analysen, insbesondere bei großen Datenmengen.

Während zeilenorientierte Datenbanken ganze Datensätze abrufen, liest eine spaltenorientierte Datenbank nur die benötigten Spalten. Dadurch wird die Menge der übertragenen Daten reduziert, was die Speichernutzung optimiert und die Abfragegeschwindigkeit verbessert.

Ein weiterer Vorteil ist die Parallelisierung von Abfragen. Da Spalten unabhängig voneinander verarbeitet werden können, lassen sich komplexe Berechnungen effizient auf mehreren Prozessorkernen oder sogar verteilten Systemen ausführen. Dies ermöglicht eine schnelle Suche.

Suchmethoden

Milvus unterstützt viele verschiedene Sucharten, darunter:

- **ANN-Suche:** Diese Methode sucht die k nächsten Vektoren, die dem Abfragevektor am ähnlichsten sind.
- **kNN-Suche:** Die kNN-Suche findet die k ähnlichsten Vektoren zu einem Abfragevektor, um schnell ähnliche Daten zu ermitteln.
- **Filternde Suche:** Hier wird eine ANN-Suche mit zusätzlichen Filterkriterien kombiniert, sodass nur eine bestimmte Teilmenge der Daten durchsucht wird, beispielsweise nach Kategorien oder Werten.
- **Bereichssuche:** Diese Suche identifiziert Vektoren, die innerhalb eines bestimmten Abstands (Radius) um den Abfragevektor liegen. Das ist nützlich, um Daten in der Nähe eines bestimmten Punktes zu finden.
- **Hybride Suche:** Kombiniert die ANN-Suche mit mehreren Vektorfeldern, um komplexere Suchanfragen zu ermöglichen.
- **Volltextsuche:** Diese Methode durchsucht Textdaten effizient, basierend auf Algorithmen wie BM25, um relevante Ergebnisse zu liefern.
- **Neu ordnen:** Nach einer ersten Suche können die Ergebnisse neu sortiert werden, basierend auf zusätzlichen Kriterien oder einer sekundären Analyse, um die Reihenfolge zu optimieren.
- **Abrufen:** Diese Methode ermöglicht den direkten Zugriff auf Daten mithilfe ihrer Primärschlüssel. Sie ist besonders schnell, da die genaue Adresse der Daten bekannt ist.

- **Abfragen:** Daten werden basierend auf bestimmten logischen Bedingungen oder Ausdrücken abgerufen. Diese Methode erlaubt sehr spezifische und angepasste Abfragen.

In unserer Diplomarbeit verwenden wir die **ANN-Suche** um die nächsten Vektoren zu finden, die dem Abfragevektor am ähnlichsten sind.

2.3.3 Entity Framework Core

Entity Framework Core ist eine Toolbox für den Datenzugriff aus Datenquellen. Von Microsoft entwickelt, bietet es volle Integration im .NET Core System.

EF Core besitzt ORM, welches für die Abbildung der Modelle im MS SQL-Server zuständig ist. Eine weitere Technologie sind die Migrationen, welche für die Erzeugung von Datenbanken mit den Modellschemas zuständig sind. Diese Migrationen sind nach Datum sortiert. Löst man manuell ein Update der Datenbank aus, wird die neueste Migration für ein delete-and-create verwendet. [22]

Unser System verwendet eine MS SQL-Datenbank für die Speicherung von Chat-Daten. Mittels ORM können diese im Backend hinzugefügt und bearbeitet werden, ohne SQL-Befehle schreiben zu müssen.

Für Modelle können auch Eigenschaften als Konfigurationen definiert werden. Sind dies bestimmte Anforderungen, müssen diese vor der Persistierung gegeben sein. Andernfalls wird EF Core einen Fehler werfen.

```
1  public class ChatConfiguration : IEntityTypeConfiguration<Chat>
2  {
3      public void Configure(EntityTypeBuilder<Chat> builder)
4      {
5          builder.HasKey(c=>c.Id);
6
7          builder.Property(c => c.Name)
8              .IsRequired()
9              .HasMaxLength(200);
10
11         builder.Property(c => c.UserGUID)
12             .IsRequired();
13
14         builder.HasMany(c => c.QueryItems);
```

```
15      }
16  }
```

Im zuvor angeführten Codebeispiel ist das Modell Chat ersichtlich. Dieses hat einen Schlüssel definiert, benötigt einen Namen mit maximal 200 Zeichen und eine UserGUID. Der Chat kann eine Liste mit mehreren QueryItems enthalten.

2.4 Verwendete Entwicklungssysteme

2.4.1 Visual Studio

Für die Entwicklung aller .NET Anwendungen wird die Entwicklungsumgebung Visual Studio (Version 2022) verwendet. Diese IDE ist ein Produkt von Microsoft und bietet für dessen Technologien (ASP.NET Core, EF Core, ...) volle Unterstützung. Visual Studio beinhaltet auch einen „Server-Explorer“, welcher für die direkte Bearbeitung von Datenbanken verwendet werden kann. Weiters sind einige Debug-Tools vorhanden, um Programme während ihrer Laufzeit auf Fehler zu prüfen.

Visual Studio verwendet den Package Manager NuGet. Einige der importierten Pakete sind z.B. ElasticSearch.Net oder Microsoft.ML. In unserem System verwenden wir auch ein Paket des Unternehmens ITPRO für die Authentifizierung von Benutzern.

In Visual Studio gibt es ebenfalls die Möglichkeit Repositorys und dessen Branches zu verwalten.

2.4.2 Visual Studio Code

Visual Studio Code ist ein kostenloser Quelltexteditor, welcher von Microsoft entwickelt wird. Die Einfachheit sowie die Möglichkeit plattformübergreifend arbeiten zu können, machen Visual Studio Code neben der hohen Kompatibilität für viele Programmiersprachen und sonstige IT-Tätigkeiten durch die unzähligen verfügbaren Extension zu einem unabdingbaren Programm bei der Entwicklung von Softwareprojekten.

Abbildung 16: VS Code Logo seit 2019 [23]

Extensions

Visual Studio Code bietet zahlreiche Extensions, die die Arbeit vereinfachen. Einige Extensions wurden von Microsoft, die meisten aber von der Community entwickelt. Im Folgenden werden die verwendeten Extensions aufgezählt und beschrieben.

- **Git Graph:** Git Graph ist eine von der Community entwickelte Erweiterung, die Git-Projekte übersichtlich darstellt und verschiedene Git-Funktionen ausführen kann. Git Graph wird von dem User *mhutchie* seit 2019 entwickelt.
  GitHub Repository: Git Graph
- **Prettier:** Prettier hilft dem Entwickler dabei, Code und JSON-Strings zu formatieren, um sie lesbarer zu machen. Entwickelt wird diese Extension von prettier.io und ist seit 2017 verfügbar.
  GitHub Repository: Prettier
- **Python:** Die seit 2016 verfügbare Python-Erweiterung von Microsoft unterstützt Python-Entwickler bestmöglich. Bei der Installation werden die Extensions Pylance und Python Debugger mitinstalliert und integriert. Pylance bietet kontextbezogene Codevervollständigungen und der Python Debugger ermöglicht die Fehlersuche in Python-Scripts. Andere nützliche Funktionen sind das Formatieren von Code, Code-Navigation und das Refactoring.
  GitHub Repository: Python
- **Rainbow CSV:** Rainbow CSV ist eine Software, die es erlaubt, Spalten in CSV-Dateien zu highlighten, um Werte leichter zuordnen zu können. Ein Kompatibilitätscheck überprüft beispielsweise Anführungszeichen und Zeilenanzahlen. Eine weitere nützliche Funktion ist die Abfragesprache „RBQL“. Rainbow CSV wurde von *mechatroner* 2017 entwickelt.
  GitHub Repository: Rainbow CSV
- **Docker:** Die Docker Extension ist seit 2015 verfügbar und wird ebenfalls von Microsoft entwickelt. Neben der grafischen Übersicht aller Container können auch Docker Networks und Volumes angezeigt werden. Für Dockerfiles und Docker-Compose-Dateien gibt es IntelliSense, um die Erstellung zu erleichtern.
  GitHub Repository: Docker

Website: Visual Studio Code

[23] [24]

2.4.3 Webstorm

WebStorm ist eine von JetBrains entwickelte Entwicklungsumgebung (IDE) für JavaScript, TypeScript und Frameworks wie Angular. Wir verwenden WebStorm, da diese Entwicklungsumgebung am Besten für das Entwickeln in Angular geeignet ist. Durch die hervorragende Autovervollständigung und der integrierten Git-Unterstützung, die uns das Mergen vom Code erleichtert, ist WebStorm die ideale Entwicklungsumgebung für uns. [25]

Abbildung 17: Webstorm Logo

2.4.4 Postman

Postman ist eine Desktopanwendung, um alle Arten von APIs während der Entwicklung testen zu können. Die Postman Workspaces ermöglichen es alleine, im Team oder sogar öffentlich zusammenzuarbeiten. In jedem Workspace gibt es die Möglichkeit, verschiedene Repositorys für unterschiedliche APIs anzulegen und zu verwalten. Neben der Möglichkeit zusammenzuarbeiten ist eine der nützlichsten Funktionen von Postman die Möglichkeit Variablen zu definieren, die im Aufruf, in der URL oder auch im Request-Body verwendet werden können, was die Test um einiges effizienter und wiederverwendbarer macht.

Abbildung 18: Postman Logo [26]

Des Weiteren bietet Postman die Möglichkeit APIs zu dokumentieren, was wiederum die Zusammenarbeit um einiges erleichtert.

Website: Postman

2.5 Verwendete Bibliotheken und Plugins

2.5.1 ExifTool

ExifTool ist eine von Phil Harvey entwickelte Perl Bibliothek, die auch als Konsolenapplikation auf fast allen gängigen Plattformen verfügbar ist. Das ExifTool wurde dafür gemacht, Metadaten aus allen möglichen Dateitypen und den zusammenhängenden Metadaten-Typen zu lesen, zu schreiben bzw. zu bearbeiten oder sogar zu vergleichen. Es stellt sich als beste Alternative für die Metadatenextraktion heraus, da es nicht nur für die gängigsten Dateiformate wie PDF, DOC, DOCX, ODT, TXT usw. immer alle verfügbaren Metadaten in einem weiterverarbeitbaren Format zurückgibt, sondern für insgesamt mehr als 300 verschiedenen Dateitypen. Darunter auch JPEG und PNG Dateien, sowie RAW-Bild-Dateien, aus denen oft sogar die genauen Aufnahmedaten und Herstellerangaben der Digitalkameras von bekannten Herstellern wie Canon, Sony, Panasonic, DJI, GoPro uvm. extrahiert werden können. Dieses Tool wurde durch die oben genannten Vorteile anderen Möglichkeiten, wie der Metadatenextraktion durch Tika vorgezogen.

```
> exiftool -h -canon t/images/Canon.jpg
```

File Name	Canon.jpg
Camera Model Name	Canon EOS DIGITAL REBEL
Date/Time Original	2003:12:04 06:46:52
Shooting Mode	Bulb
Shutter Speed	4
Aperture	14.0
Metering Mode	Center-weighted average
Exposure Compensation	0
ISO	100
Lens	18.0 - 55.0 mm
Focal Length	34.0 mm
Image Size	8x8
Quality	RAW
Flash	No Flash
White Balance	Auto
Focus Mode	Manual Focus (3)
Contrast	+1
Sharpness	+1
Saturation	+1
Color Tone	Normal
Color Space	sRGB
File Size	2.6 kB
File Number	118-1861
Drive Mode	Continuous Shooting
Owner Name	Phil Harvey
Serial Number	0560018150

Abbildung 19: ExifTool Beispiel [27]

Im Bild ist ein Beispielaufruf des ExifTools zu sehen, welcher alle Canon spezifischen Metadaten in der Konsole ausgibt.

Für genauere Informationen besuchen Sie die Website von ExifTool.

Website: exiftool.org

2.5.2 LangChain

LangChain wurde als .Net Bibliothek eingebunden, um die extrahierten Inhalte aus den Dokumenten in mehrere sogenannte „Chunks“ aufzuteilen. Ein Chunk ist sozusagen ein Textabschnitt mit einer maximalen Anzahl an Zeichen. Das ist wichtig, da große Dokumente nicht als Ganzes an das LLM zur Weiterverarbeitung gesendet werden können, da die Übertragungsgröße meistens

begrenzt ist. Wichtig ist aber, dass die Originaltexte nicht willkürlich in eben solche Chunks aufgeteilt werden, zum Beispiel immer nach genau 4000 Zeichen, da dadurch Zusammenhänge und eventuell wichtige Informationen verloren gehen können. Um das zu beheben wurde der Recursive Character Text Splitter von LangChain verwendet, welcher immer versucht Textabschnitte so gut wie möglich beisammen zu halten und nur bei Absätzen oder am Satzende einen Split zu machen, damit keine Zusammenhänge verloren gehen.

RecursiveCharacterTextSplitter

Im Gegensatz zu normalen Character Splittern wie in Abbildung 20, bei denen immer nach einer fixen Anzahl an Zeichen der Text aufgeteilt wird und es somit, wie im Beispiel zu sehen, dazu kommt, dass mitten im Satz oder sogar inmitten eines Wortes ein Split gemacht wird, versucht der Recursive Character Text Splitter dies so gut wie möglich.

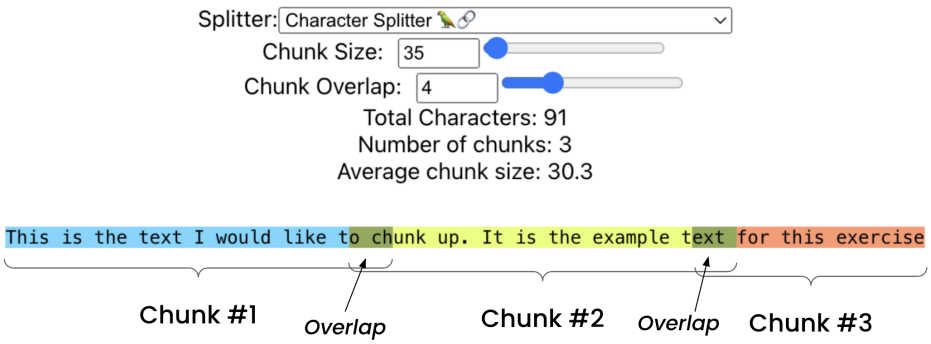


Abbildung 20: Character Splitter

Wie in Abbildung 21 zu sehen, teilt der Recursive Character Text Splitter den Text nicht immer nach genau 469 Zeichen, sondern versucht so viele Absätze wie möglich in einen Chunk zu verpacken. Haben also mehrere Absätze zusammen weniger als 469 Zeichen können diese zu einem Chunk zusammengefasst werden. Ist ein Absatz aber größer als 469 Zeichen muss dieser aufgeteilt werden, damit er in einem Chunk Platz hat. Dabei wird dann versucht, immer nach Satzenden zu teilen. Ist das aber immer noch nicht möglich, da ein Satz mehr als 469 Zeichen hat, wird erst dann ein Satz selbst aufgeteilt, wobei auch wieder zwischen den Wörtern geteilt wird. Ein Teilen innerhalb eines Wortes ist sehr unwahrscheinlich, da die Chunkgröße meist um ein Vielfaches größer ist als die Länge einzelner Worte, da ansonsten gar keine Zusammenhänge mehr in den Chunks wären.

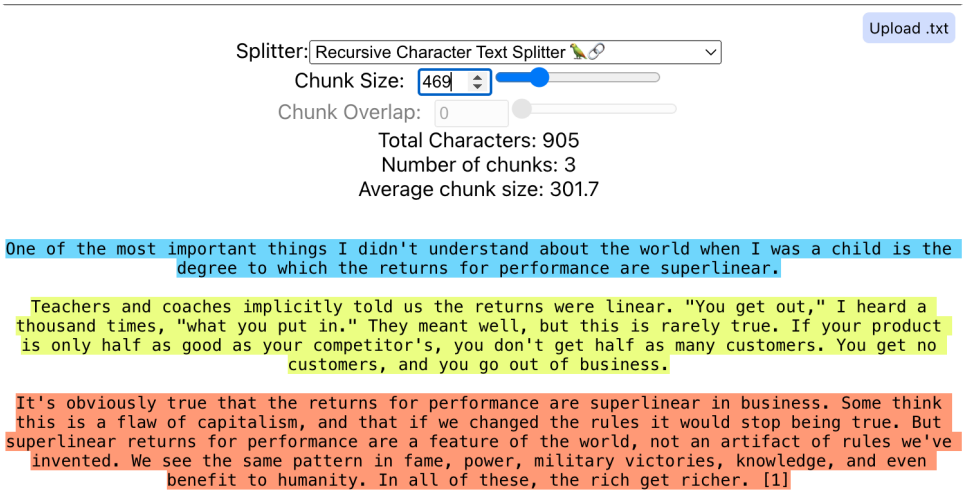


Abbildung 21: Recursive Character Splitter

Website: LangChain

2.5.3 Pandas

Diese Open-Source Python Bibliothek ist ein weit verbreitetes, leistungsstarkes Tool, um Daten in Python tabellarisch-strukturiert zum Beispiel in DataFrames zu speichern sowie abzufragen und zu analysieren. Gerade in Bereichen Data-Science und künstliche Intelligenz wird Pandas gerne verwendet, da es auch nützliche Funktionen wie das Bereinigen von fehlerhaften Daten oder auch einfache Visualisierungen ermöglicht.

In unserem Projekt wurde Pandas dazu verwendet, die Ergebnisse der Analyzers zwischenzuspeichern, um später daraus Analysen erstellen zu können.

Website: Pandas

2.5.4 Seaborn

Die Datenvisualisierung-Bibliothek für Python basiert auf der Bibliothek Matplotlib und ermöglicht es attraktive und informative Datenvisualisierungen zu erstellen. Seaborn wird oft in Kombination mit Pandas verwendet und bietet dutzende unterschiedliche Diagrammtypen, die nicht nur farblich an die eignen Ansprüche angepasst werden können. Alleine in der Beispielgalerie auf der Webseite von Seaborn werden knapp 50 unterschiedliche Beispiele gezeigt, wie Seaborn verwendet werden kann.

In diesem Projekt wird Seaborn im Analyzer bei den Auswertungen der Analysen verwendet, um die Ergebnisse ansprechend zu visualisieren, um sie schnell verstehen zu können.

Website: seaborn.pydata.org

2.5.5 Requests

Requests ist ebenfalls eine Bibliothek für Python. Durch die gute Dokumentation und einfach gehaltene Syntax, welche es erlaubt http-Requests in nur einer Zeile zu implementieren ist diese Bibliothek sehr einsteigerfreundlich.

Im Projekt wurde Requests ebenfalls im Analyzer verwendet. Dort diente sie dazu, die API-Anfragen aus dem Frontend an das Backend zu simulieren, um die Richtigkeit sowie Geschwindigkeit der Antworten zu überprüfen.

Library Dokumentation: Requests

2.5.6 Tkinter

Die Python Bibliothek Tkinter ist dafür bekannt, es zu ermöglichen, schnell einfache GUI-Anwendungen in Python zu entwickeln. In die Tkinter Benutzeroberfläche lassen sich viele Widgets wie Buttons, Textfelder und Labels einfügen, welche es ermöglichen, schnell einen Prototypen für diverse Projekte implementieren zu können.

Im Projekt wurde Tkinter dazu verwendet, um eine kleine Applikation zu implementieren, die Analysen der verschiedenen Abfragevarianten auf die Datenbanken erfasst und strukturiert zu speichern, um später, auch über diese Anwendung, Analysen zu erzeugen.

Siehe mehr unter Analyzer.

Library Dokumentation: tkinter

2.5.7 Tika

Apache Tika ist ein in Java geschriebenes Tool, um Metadaten und Text aus über tausend verschiedenen Dateitypen zu extrahieren. Darunter viele oft verwendete Dateitypen wie DOC, DOCX, ODT, PDF uvm. aber auch Text aus Bildern kann mittels OCR extrahiert werden. Apache Tika gibt es als Java Bibliothek, sowie auch als REST-Server Variante, die sich sehr gut unter Docker ausführen lässt, um ihn so in allen gängigen Programmiersprachen verwenden zu können. Im Projekt wurde nur die Textextraktionsfunktion verwendet.

Website: tika.apache.org

2.5.8 Angular Material

Angular Material ist eine von Google entwickelte UI-Komponenten Bibliothek, die viele verschiedene vorgefertigte und anpassbare Komponenten und Features zur Verfügung stellt, die das Entwickeln von gut aussehenden und responsiven Seiten vereinfachen.

Angular Material Features

- **Materialdesignkomponente:** Angular Material bietet viele verschiedene Komponenten an, die schnell und einfach implementiert werden können.
- **Thema und Styling:** Angular Material bietet viele verschiedene Themen-Funktionen an. Mit den vorgefertigten Themen und den flexiblen Optionen, ist es möglich, schnell ein einheitliches Brandig zu erstellen.
- **Responsive Design:** Alle Angular-Material-Komponente sind responsiv. Damit ist sichergestellt, dass die Anwendung auf verschiedenen Bildschirmgrößen funktioniert und gut aussieht.
- **Integrierte Animationen:** Angular Material stellt integrierte Animationen zur Verfügung, die fließende Übergänge ermöglicht. Diese Animationen sind in den Animationsfunktionen von Angular integriert, deshalb eine schnelle und einfache Implementierung der Animationen möglich ist.

[28]

2.5.9 NGXS

NGXS ist eine Angular-Bibliothek für State Management. Dadurch ist es möglich, den Zustand einer Angular-Anwendung zentral zu verwalten. In unserem Projekt verwenden wir NGXS, da die Implementierung des State Management mit NGXS einfacher ist, als mit anderen State Management-Bibliotheken.

NGXS basiert auf vier Hauptkonzepten, die die Verwaltung des States ermöglichen. Um die Hauptkonzepte besser zu verstehen, erfolgt die Erklärung gemeinsam mit einem kleinen Code-Beispiel für eine einfache Zooverwaltung. Einen Überblick darüber, wie die verschiedenen Komponenten in NGXS zusammenarbeiten, zeigt die Abbildung 22.

Actions

Actions sind Ereignisse, die eine Änderung im Zustand der Anwendung auslösen. Sie können als Befehle betrachtet werden, die bestimmte Operationen im State ausführen sollen.

```
export class AddAnimal {
  static readonly type = '[Zoo] Add Animal';
  constructor(public name: string) {}
}
```

Jede Action besitzt eine eindeutige Typenbezeichnung. In diesem Beispiel erfordert die Action einen String-Parameter, der den Namen des Tieres enthält. Beim Dispatchen dieser Action wird ein neues Tier zum State hinzugefügt.

State

Ein State ist eine Klasse, die den Anwendungszustand speichert und auf Actions reagiert, um den Zustand zu ändern.

```
interface AnimalStateModel {
  animals: string[];
}

@State<AnimalStateModel>({
  name: 'animals',
  defaults: {
    animals: []
  }
})
export class AnimalState {
  @Action(AddAnimal)
  addAnimal(ctx: StateContext<AnimalStateModel>, action: AddAnimal) {
    const state = ctx.getState();
    ctx.setState({
      animals: [...state.animals, action.name]
    });
}
```

Hier wird das State Model AnimalStateModel definiert, das eine Liste von Tieren speichert. Die Action AddAnimal wird in der Methode addAnimal verarbeitet, indem sie das neue Tier zur

Liste hinzufügt.

Store

Der Store ist das zentrale Element des NGXS-Systems. Er ermöglicht das Dispatchen von Actions und den Zugriff auf den State, ohne dass Komponenten direkt voneinander abhängig sind.

```
constructor(private store: Store) {}
addAnimal() {
    this.store.dispatch(new AddAnimal("New Animal"));
}
```

Der Store wird in der Komponente eingebunden. Die Methode dispatch sendet die AddAnimal-Action an den State, wodurch ein neues Tier im State hinzugefügt wird.

Select

Mit Select können Werte direkt aus dem State abgerufen werden. Dadurch bleibt die Komponente immer synchron mit den aktuell gespeicherten Daten.

Es gibt mehrere Methoden zur Selektion in NGXS. Eine häufig verwendete Methode wird im folgenden Beispiel beschrieben.

```
@Select(AnimalState) animals$: Observable<string[]>;
```

Durch den @Select-Dekorator wird auf die Liste der Tiere im State zugegriffen. Die Variable animals$ ist ein Observable, das automatisch aktualisiert wird, sobald sich der Zustand ändert. [29]

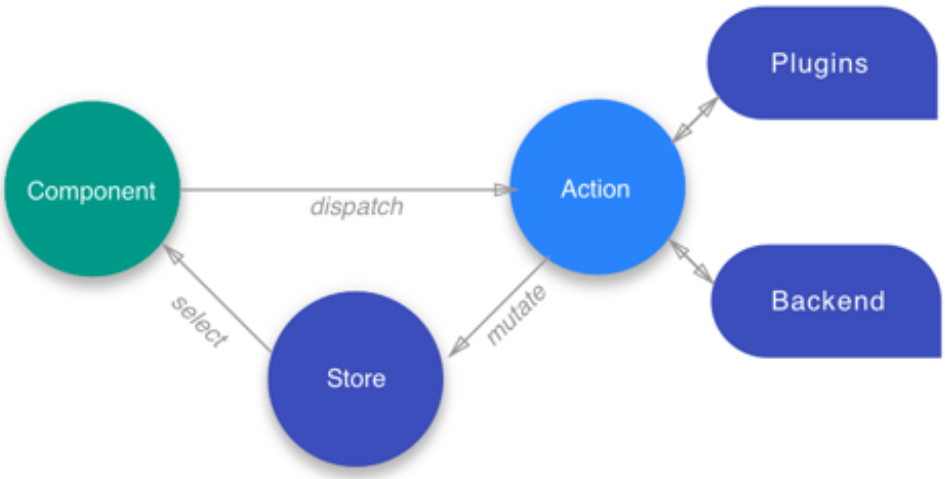


Abbildung 22: NgXS Overview [30]

2.5.10 OpenAI API

OpenAI API ist eine kommerziell zugängliche REST-API. Mit dieser Schnittstelle ist es Programmen möglich, die Funktionen von ChatGPT zu nutzen. Das Unternehmen ITPRO stellt uns einen API-Key zur Verfügung, mit welchem unser System Anfragen an die AI stellen kann. Die API nutzt ein nutzungsbasiertes Abrechnungsmodell. Die Anfragen werden aufgezeichnet, die Anzahl der verbrauchten Tokens berechnet und am Ende des Monats dementsprechend bezahlt. [31]

Für unser System nutzen wir die OpenAI API an zwei Stellen:

1. Die Suchdatenbanken benötigen bestimmte Objekte, um nach einem richtigen Dokument zu suchen. Bei Elasticsearch wird dabei ein semantisch korrektes JSON-Objekt gefordert, bei Milvus ein Vector-Embedding. Die Generierung dieser Objekte übernimmt die OpenAI-API.

2. Das Backend erhält von den Datenbanken ein Dokument als Response. Um daraus eine sprachlich natürliche Antwort zu erstellen, wird ebenfalls die API verwendet.

2.5.11 Elastic.Clients.Elasticsearch

Dies ist der offizielle .NET Client von Elasticsearch. Er stellt stark typisierte Funktionen für den Zugriff auf ElasticSearch-Indizes zur Verfügung. Der Client benötigt für den Transport eine Connection, welche den Index und den dazugehörigen API-Key beinhalten muss.

Da unser System mit JSON-Queries arbeitet verwenden wir die Funktion ElasticsearchClient.Transport.RequestAsync<StringResponse>(). Die JSON-Objekte müssen bei jeder Anfrage in korrekter Form übergeben werden, da die Typisierung übergangen wird.

2.6 Verworfene Optionen

2.6.1 Ollama

Ollama bietet mehrere Large Language Model an, mit dem Zweck, diese lokal auf eigenen Rechnern zu hosten. Dies hätte den Vorteil keine Verbindung zu OpenAI herstellen zu müssen sowie dorthin auch keine Daten weiterzugeben. Die Modelle könnten einerseits in der Antwortgenerierung als auch in der Querygenerierung für die Suchdatenbanken eingesetzt werden.

Ein großes Problem dabei ist, dass ein solches Large Language Model sehr viel Prozessorkapazität benötigt. Mit unseren Rechnern konnten wir niemals zufriedenstellende Ergebnisse erreichen, das Generieren von Antworten benötigte teilweise bis zu 20 Minuten. Ein weiteres Problem sind die generierten Embeddings für die Vektorsuche. Dafür gibt es ein eigenes Vektormodell, welches qualitativ weit von OpenAI abgehängt wird. Es lassen sich keine guten Ergebnisse damit erzielen, die Suche schlägt meistens fehl.

2.6.2 ML .NET

Zu Beginn wurden im Projekt Vektor-Embeddings mithilfe von ML.NET erzeugt, was anfänglich auch sehr sinnvoll war, da sie auf lokalen Systemen lief. Das Modell konnten direkt auf unseren Systemen ausgeführt werden.

Abbildung 23: ML .NET-Logo [32]

Allerdings stellten wir nach Testen von Datenbankabfragen fest, dass diese Vektoren nicht die Qualität bei den Vektoren boten, die wir uns gewünscht haben. Insbesondere bei komplexeren Anfragen lieferte das Modell immer wieder falsche Ergebnisse.

Deshalb haben wir einen Blick auf OpenAI (text-embedding-ada-002) geworfen, welches eine API-basierte Lösung bereitstellt, die in ihrer Anwendung deutlich leistungsfähiger ist, auch wenn es etwas kosten. Die Modelle sind dort bereits optimiert und werden immer weiterentwickelt, ohne dass wir selbst für Updates und Anpassungen sorgen müssten. So konnten wir die Qualität der Suchergebnisse entscheidend erhöhen.

2.6.3 FSCrawler

In den ersten Versionen dieses Projekts wurde zum Indizieren von Dokumenten das Tool FSCrawler verwendet. Grundlegende Funktionalitäten konnten sehr einfach und schnell mit dem FSCrawler umgesetzt werden, da es möglich ist, ihn gemeinsam mit Elasticsearch in einem Docker Compose zu starten. Dafür gibt es bereits Anleitungen, um mit den Standardeinstellungen zum ersten Erfolg zu kommen: FSCrawler and Elasticsearch using Docker Compose

Nachdem erste Versuche ziemlich vielversprechend aussahen, stießen wir bald auf ein Problem. FSCrawler indiziert jede Datei als eigenes Dokument in Elasticsearch, was auf Datenbankebene zwar sinnvoll und auch so angedacht wäre. Im weiteren Programmablauf stellt dies aber leider ein schwerwiegendes Problem dar. Durch dieses Verfahren wächst mit steigender Originaldatei-Größe auch die Größe des in Elasticsearch indizierten Dokuments. Ab einer gewissen Datei- bzw. Dokumentengröße ist dieses Verhalten für unser Programm leider nicht mehr tragbar, da die Ergebnisdokumente aus dem Query erneut an das LLM gesendet werden sollen. Beim Senden an das LLM gibt es aber eine Maximalanzahl an Zeichen bzw. eine maximale Übertragungsgröße, welche bei mehrseitigen PDF-Dokumenten schnell erreicht ist und das Ziel der Anwendung aber sein soll, Antworten auf Fragen in einer großen Menge von großen Dokumenten zu finden.

Ein weiterer Punkt, welcher schlussendlich zum Verwurf von FSCrawler geführt hat, war die Integration einer zweiten Datenbank. Da eine Vektordatenbank als zweite Datenbank in unser System hinzugefügt werden sollte, musste sowieso eine Möglichkeit gefunden werden, die Daten auch in diese Datenbank zu speichern, was mit FSCrawler aber nicht wirklich möglich ist.

Aufgrund dieser Argumente wurde beschlossen, die Datenbeschaffung mittels FSCrawler zu verwerfen und auf Alternativen zu setzen, was schlussendlich dazu geführt hat, die Datenbeschaffung datenbankunabhängig im Backend eigenständig zu implementieren. Genaueres siehe im Kapitel Datenbeschaffung

Website: FSCrawler

2.7 Verwendete Konzepte

2.7.1 SSO

SSO (Single Sign-On) ist eine Authentifizierungsmethode, die es ermöglicht, mit einem Login Zugriff auf mehrere Anwendungen oder Websites zu haben. Dadurch muss sich der Benutzer nur einen Anmeldesatz merken und sich nicht für jede Anwendung separat anmelden.

Funktion von SSO

Beim Anmeldeversuch eines Benutzers mit SSO wird immer folgender Prozess ausgeführt:

- Als erstes überprüft SSO, ob der Benutzer bereits authentifiziert wurde. Wenn der Benutzer schon authentifiziert ist, wird ihm der Zugang zum Dienst gewehrt.
- Wenn der Benutzer noch nicht authentifiziert ist, wird er zur Anmeldeseite weitergeleitet.
- Bei der Anmeldeseite gibt der Benutzer seine Anmeldedaten ein, die meistens aus Benutzername und Passwort bestehen.
- Zur Authentifizierung prüft das SSO-System die Anmeldeinformationen des Benutzers mit einem vertrauenswürdigen Identitätsanbieter.
- Nachdem sich der Benutzer erfolgreich angemeldet hat, kann der Benutzer auf alle Dienste zugreifen, die dasselbe SSO-System verwenden, ohne seine Anmeldeinformationen erneut einzugeben.

Vorteile

- **Benutzerfreundlichkeit:** Durch SSO muss sich der Benutzer nicht mehr so viele Anmeldedaten merken und der Benutzer hat nach einem Login Zugriff auf mehrere Anwendungen.
- **Produktiver:** Da sich der Benutzer nicht mehr bei jeder Anwendung separat anmelden muss, spart sich der Benutzer Zeit und Aufwand.
- **Einfachere Administration:** Die Identitäten und Berechtigungen der Benutzer können durch SSO besser und zentralisiert verwaltet werden.

[33] [34]

2.7.2 Text Embedding

text-embedding-ada-002 ist ein Modell von OpenAI, das verwendet wird, um Texte in numerische Vektoren umzuwandeln. Diese Vektoren bestehen aus Zahlenwerten, die den Inhalt eines Textes in numerische Werte abbilden und es ermöglichen, semantische Ähnlichkeitssuche zwischen verschiedenen Texten zu erkennen. Ziel ist es, die Bedeutung und den Inhalt von Texten in einer Form darzustellen, die von Maschinen besser verarbeitet werden können. [35]

Funktionsweise des Modells

Das Modell analysiert den eingegebenen Text und erstellt auf Basis von Mustern einen Vektor mit 1536 Dimensionen. Jede Dimension ist eine Zahl, die einen Teil des Textes beschreibt. Ähnliche Texte erzeugen ähnliche Vektoren, was das Modell ideal für eine Textsuche macht.

Verwendung von Text-Embedding in Milvus

1. **Textverarbeitung:** Ein Text wird an das Modell übergeben.
2. **Vektorisierung:** text-embedding-ada-002 erzeugt einen Vektor mit 1536 Dimensionen.
3. **Speicherung in Milvus:** Der Vektor wird in der Vektor-Datenbank gespeichert.
4. **Ähnlichkeitssuche:** Bei der Suche berechnet Milvus die Ähnlichkeit der gespeicherten Vektoren mit dem Suchtext, der ebenfalls vektorisiert wird.

2.7.3 Lexikalische Suche

Die lexikalische Suche analysiert Texte, indem sie Wörter standardisiert, vergleicht und Muster erkennt, um Dokumente zu finden, die auf übereinstimmenden Suchbegriffen basieren. Wie mithilfe einer umgekehrten Indexstruktur lexikalisch gesucht werden kann, wird hier beschrieben: Umgekehrte Indexstruktur

2.7.4 Vektorsuche

Die Vektorsuche ist eine Methode, um Inhalte in einer Datenbank nach einer numerischen Repräsentation zu finden. Im Gegensatz zur klassischen Suche muss man bei der Vektorsuche die Wörter nicht direkt miteinander verglichen werden. Man analysiert bei der Vektorsuche die Bedeutung und Zusammenhänge der Daten. Das ermöglicht eine viel genauere und flexiblere

Suche, da die Eingabe nicht 100-Prozent mit den Daten in der Datenbank übereinstimmen muss, sondern auch ähnliche oder verwandte Inhalte gefunden werden können.

Das funktioniert, weil die Daten wie Texte, Bilder, oder andere Inhalte in mathematische Vektoren umgewandelt werden. Auf diese Weise wird alles in einem mehrdimensionalen Raum dargestellt. Ein Vektor ist eine Zahlenreihe, die die wesentlichen Merkmale eines Inhalts in einem mehrdimensionalen Raum darstellt. Ähnlichen Inhalte haben ähnlich Vektoren.

Wenn eine Suchanfrage gestellt wird, verwandelt Milvus den Suchtext auch in einen Vektor um und sucht sie in den Daten, die in der Datenbank gespeichert sind. Milvus vergleicht den Suchvektor mit den gespeicherten Vektoren und gibt das beste Ergebnis zurück. Dadurch können nicht nur exakte Übereinstimmungen gefunden werden, sondern auch inhaltliche verwandte Ergebnisse. [36]

Wie haben wir im Projekt die Vektorsuche eingebaut?

1. Im Angular-Frontend kann eine Frage eingegeben werden
2. Im Backend werden von der Suchanfrage die wichtigsten Begriffe extrahiert
3. Die Begriffe werden in Vektoren umgewandelt
4. Der Chunk, der am besten zur Suchanfrage passt wird zurückgeben

2.7.5 Retrieval-Augmented Generation

Retrieval-Augmented Generation ist eine Technik im Bereich der Artificial Intelligence. LLMs werden mit internem, spezifischem Wissen kombiniert. Diese Modelle neigen dazu, ihre Antworten sehr oberflächlich zu halten. Die arbeitenden neuronalen Netze verfügen über ein sehr breites Wissen und ahmen nur das generelle Pattern einer menschlichen Antwort nach.

Die RAG-Technik ist erstmals 2020 von Patrick Lewis beschrieben worden und das Ziel ist es tiefer und spezifischer mit bestimmten Daten arbeiten zu können. Die Ressourcen werden in externen Datenbanken gespeichert, Wissensbasis genannt. Ein Controller verbindet diesen Speicher mit Abfragen an ein LLM und man erhält relevantere Antworten. Anwendung finden diese Systeme z.B. in der Medizintechnik bei der Suche nach richtigen Behandlungen oder in Rechtsabteilungen, um die relevanten Gesetze für einen bestimmten Fall zu finden. [37])

In unserem Projekt bildet das Backend den Controller. Die Elasticsearch und Milvus Datenbanken sind die Wissensbasis und die OpenAI-API ist das verwendete LLM. Wird in einem Chat eine Frage gestellt, werden die Daten wie folgt verarbeitet:

1. Die Frage wird vom Frontend an die API im Backend versendet.
2. Die Frage wird gemeinsam mit einem vordefinierten Text umformatiert und an die OpenAI-API gesendet.
3. Von der OpenAI-API erhält unser System einen JSON-Body (Elasticsearch) oder ein Embedding (Milvus).
4. Das erhaltene Objekt wird an die **ausgewählte Datenbank** versendet und auf eine Response gewartet.
5. Die ausgewählte Datenbank sendet die Response zurück. In dessen Body sind mehrere Ausschnitte aus Dokumenten vorhanden.
6. Das Backend verwendet den Ausschnitt mit der besten **Trefferquote** und sendet dies, gemeinsam mit der ursprünglichen Frage, wieder an die OpenAI-API, um eine **natürliche Antwort** zu erstellen.
7. Hat unser System eine natürliche Antwort erhalten, kann diese, gemeinsam mit **Metadaten** des Dokuments, an den Client wieder zurückgesendet werden.

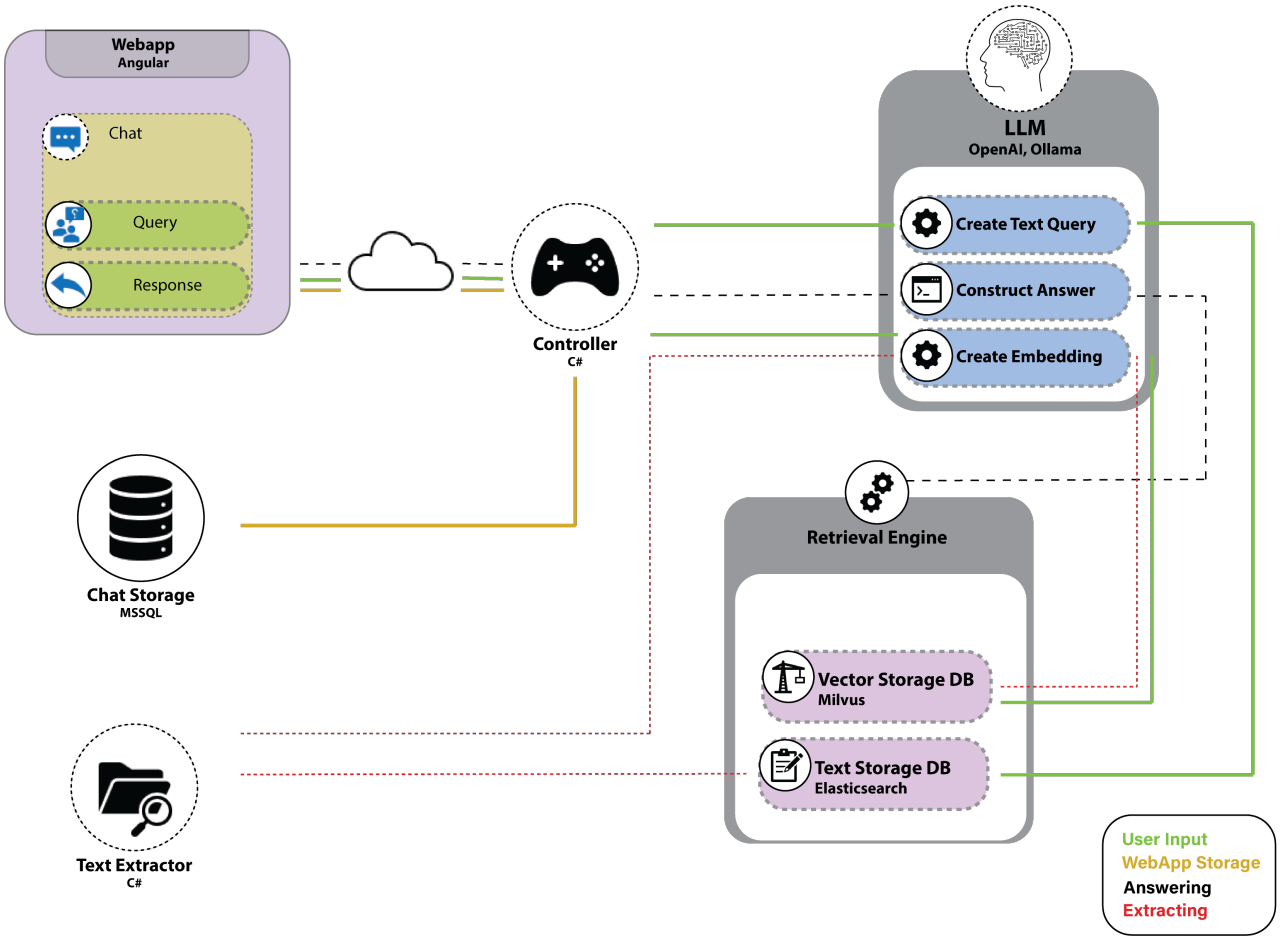


Abbildung 24: Visualisierung unseres RAG-Systems

2.7.6 ASP .NET Core Middleware Pipeline

ASP .NET Core Webanwendungen benutzen eine Middleware Pipeline, um Applikationen aufzubauen. Jede Anfrage wird von Komponente zu Komponente übergeben und eigene Logik ausgeführt. Die eigene Geschäftslogik, oft im MVC-Pattern designt, bildet das Ende der Pipeline. Von dort ausgehend werden Antworten wieder über alle Komponenten zurückgegeben. [38]

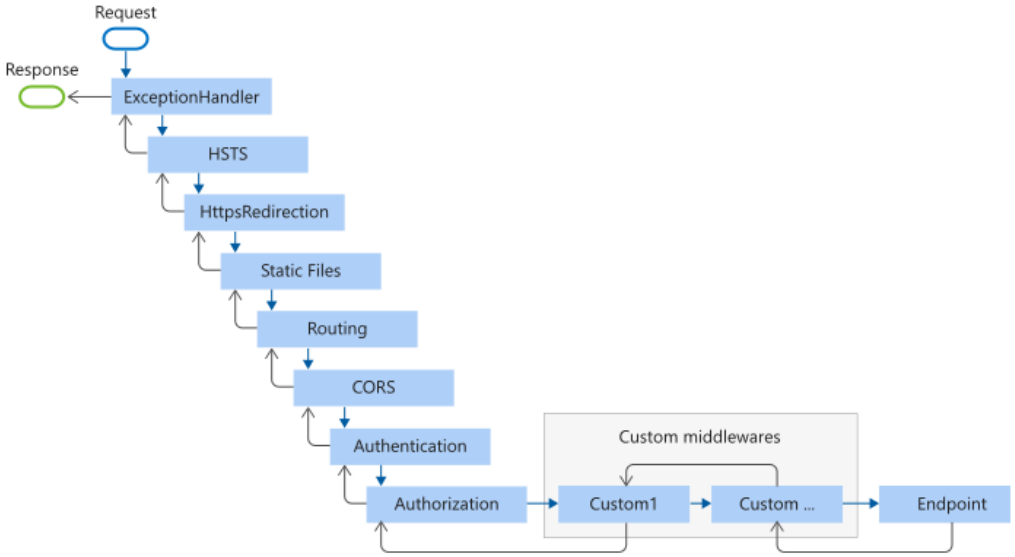


Abbildung 25: ASP .NET Core Middleware - Order of execution [38]

Die Registrierung der einzelnen Middleware-Komponenten ist in der Program.cs mit der **WebApplication-Entität** vorzunehmen. In folgendem Code-Beispiel werden die Sicherheitsmechanismen, Authentifizierung und Autorisierung unseres Backends, zur Pipeline hinzugefügt.

```
1     // Order is important, Authentication then Authorization
2     app.UseAuthentication();
3     app.UseAuthorization();
```

Für den Entwickler gibt es auch die Möglichkeit, von .NET Core zur Verfügung gestellte Middleware-Komponenten zu modifizieren. Die Software bietet so einen hohen Anpassungsgrad, um z.B. eigene Autorisierungsserver, eigene Datenbanken oder auch eigene CORS-Regeln zu verwenden. In folgendem Code-Beispiel wird die Authentifizierungs-Komponente der Middleware für den eigenen Nutzen modifiziert.

```
1 builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
2     .AddJwtBearer(o =>
3     {
4         o.Authority = builder.Configuration.GetValue<string>("SsoClient:Authority");
```

```
5           //erlaubte Token Typen
6           o.TokenValidationParameters.ValidTypes = new[] { "at+jwt" };
7           //Audience Validierung abschalten
8           o.TokenValidationParameters.ValidateAudience = false;
9       });
```

3 Planung und Realisierung

3.1 Projektorganisation

Die Diplomarbeit wurde im Rahmen eines Praktikums beim Auftraggeber ITPRO durchgeführt. Vor dem Praktikum wurden die Aufgaben der Teammitglieder, die Meilensteine und die Projektziele mit dem Diplomarbeitsbetreuer besprochen. Am ersten Tag des Praktikums fand noch ein Gespräch mit dem Auftraggeber statt, um das Projekt nochmal zu besprechen.

Nach Absprache mit dem Diplomarbeitsbetreuer und dem Auftraggeber wurden die Aufgaben innerhalb des Teams wie folgt verteilt:

- **Horner Sebastian:** Projektleiter, Elasticsearch Datenbank, Datenvorverarbeitung und Analysen
- **Kern Moritz:** Implementierung der Angular-Webanwendung
- **Pilgerstorfer Martin:** .NET Backend für die Kommunikation der Komponenten, Verwaltung der Chat-Datenbank
- **Wahl Tobias:** Milvus Datenbank und Analysen

Während des Praktikums fand jeden Tag am Morgen ein Meeting mit allen Mitarbeitern der ITPRO statt. In diesen Meetings berichtete jedes Teammitglied über seine Fortschritte vom Vortag und legte seine Ziele und Aufgaben bis zum nächsten Meeting fest. Da das gesamte Team gemeinsam in einem Büro arbeitete, war die Kommunikation miteinander sehr gut.

Zusätzlich wurde dem Diplomarbeitsbetreuer immer am Ende der Woche eine Nachricht über den Fortschritt der einzelnen Teammitglieder und den aktuellen Stand des Projekts gesendet. Durch die Projektorganisation war es möglich, die Meilensteine rechtzeitig abzuschließen und das Projekt fertigzustellen.

3.2 Meilensteine

Die Abbildung zeigt den Zeitplan unserer Diplomarbeit, in dem die einzelnen Meilensteine zeitlich dargestellt sind. Alle Einträge in der Grafik sind Meilensteine, die gelben Balken sind grobe Meilensteine des Projekts. Die unterschiedlichen Farben helfen dabei, die verschiedenen Schwerpunkte der Arbeit besser zu erkennen.

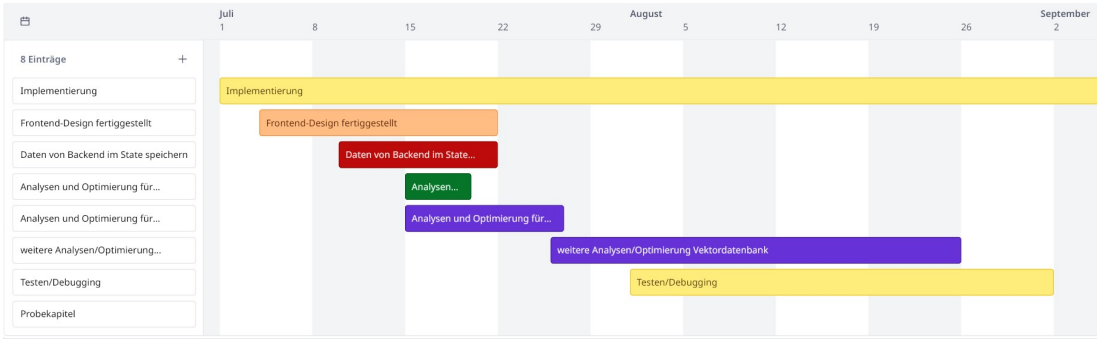


Abbildung 26: Meilensteine
[39]

Meilensteine im Projekt

- **Grobe Meilensteine (gelb)**
  - **Implementierung:** Die gesamte Entwicklungsphase der Software, in der alle Kernfunktionen umgesetzt wurden.
  - **Testen und Debugging:** Dieser Meilenstein stellt die Fehlerbehebung und Optimierungen des Projekts dar.
  - **Probekapitel schreiben:** Ein Meilenstein von September bis Dezember, bei dem das erste Kapitel geschrieben wurde.
- **Weitere Meilensteine**
  - **Frontend-Design fertiggestellt (orange)**: Fertigstellung eines modern responsiven Design
  - **Daten vom Backend im State speichern (rot):** Verwaltung und Speicherung der Daten im Backend.
  - **Datenbank-Analyse für Elasticsearch (grün):** Eine Analyse und Optimierungsphase für eine effiziente Such- und Speicherprozesse mit Elasticsearch.
  - **Analyse und Optimierung der Vektordatenbank (lila):** Ein Meilenstein zur Analyse und Optimierung der Vektordatenbank.

4 Implementierung

4.1 Technischer Überblick

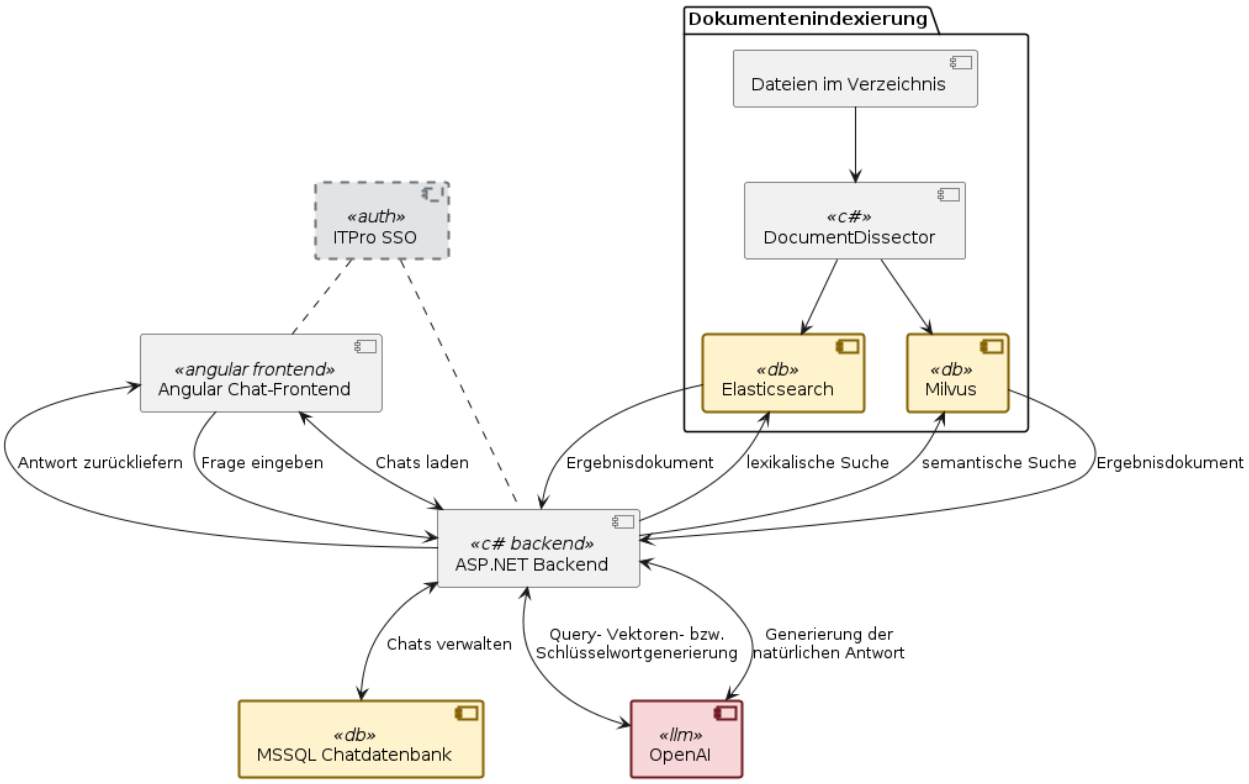


Abbildung 27: Technischer Überblick

Um den technischen Überblick visuell ansprechend darzustellen, wurde mit PlantUML ein Ablaufdiagramm erstellt. Im groben ist der Ablauf folgender:

1. Damit überhaupt abgefragt werden kann, müssen die in dem ausgewählten Verzeichnis liegenden Dokumente mithilfe des *DocumentDissectors* in die Datenbanken eingefügt werden.
2. Nach der Authentifizierung im Frontend kann in einem Chat eine Frage gestellt werden.
3. Die eingegebene Frage wird wiederum über die authentifizierte REST-API an das Backend zur Weiterverarbeitung gesendet.
4. Im Backend werden je nach Art der Abfrage entweder Schlüsselwörter für die lexikalische oder Vektoren für die semantische Suche generiert.

5. Nach der Abfragegenerierung kann auf die jeweilige Datenbank abgefragt werden.
6. Die Datenbankabfrage liefert entsprechend Dokumente.
7. Mit den Informationen aus dem Ergebnisdokument der Datenbankabfrage und der Ursprungsfrage wird durch das LLM eine natürliche Antwort generiert.
8. Die generierte Antwort wird danach ans Frontend übermittelt, in der sie angezeigt wird und in der Chatdatenbank gespeichert.

4.2 REST-API

Die REST-API unseres Systems befindet sich im Backend, welches das MVC-Pattern verwendet. Der **DocumentController** stellt nach außen alle Endpunkte dar. Die gesamte API nimmt nur JSON-Objekte als Request-Body an, welche durch ein Mapping eine Klassenbeschreibung erhalten. Wichtig zu beachten ist, dass die Verarbeitung von Daten nur für autorisierte Benutzer funktioniert. Im JSON-Header muss ein Bearer-Token vorhanden sein, welcher vom Backend bei jedem Endpunkt verifiziert wird.

Der DocumentController injiziert zwei Services:

1. IAnsweringService: Ist für die Verarbeitung einer Query (Benutzerfrage) verantwortlich (siehe Abschnitt 4.2.3).
2. IChatService: Ist für die Verwaltung der Chat-Datenbank zuständig (siehe Abschnitt 4.2.2).

Die beiden Services werden in der **Program.cs** mittels Dependency Injection registriert:

```
1  * register Services - Add Dependency Injection registration
2  */
3 builder.Services.AddScoped<IAnsweringService, AnsweringService>();
4 builder.Services.AddScoped<IChatService, ChatService>();
```

4.2.1 Authentifizierung

Die Applikation soll in die Unternehmensumgebung der ITPRO eingebaut werden können. Diese betreiben einen OAuth2-konformen Security-Service für die Autorisierung und Authentifizierung ihrer Systeme. Über das Netzwerk werden JWT-Token versendet, um ein Single-Sign-on im Frontend zu ermöglichen.

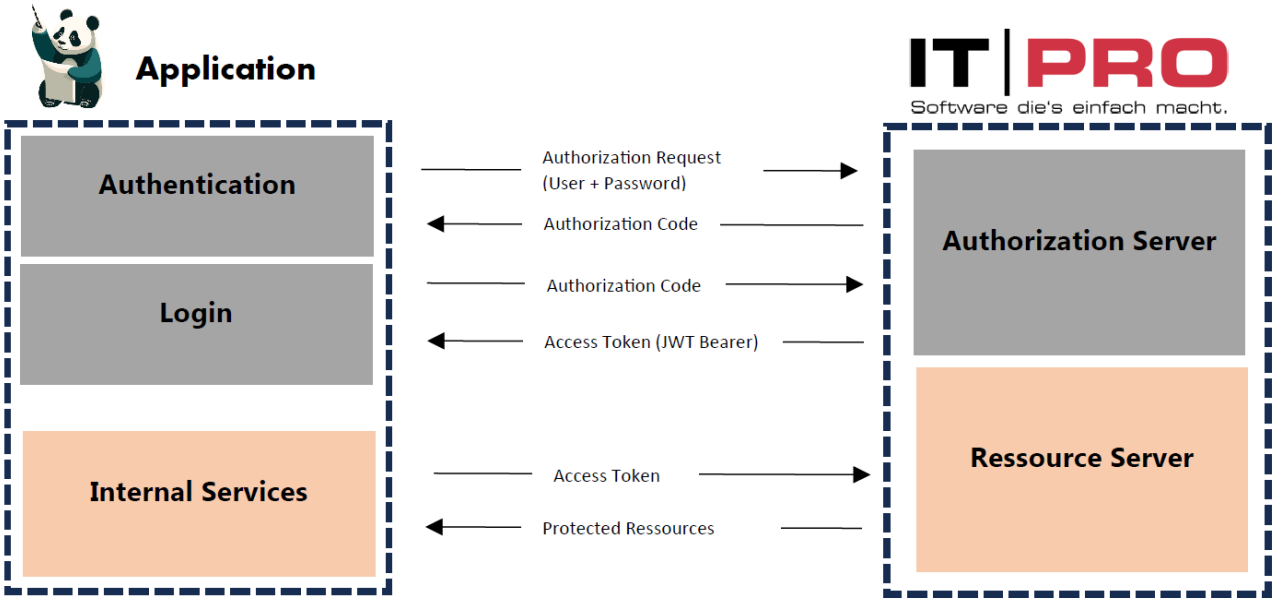


Abbildung 28: Prozess um JWT-Token und Benutzer-Ressourcen zu erhalten

Im Backend selbst muss der Client sich nicht mit Benutzernamen und Passwort anmelden. Soll eine Anfrage verarbeitet werden, muss diese im Header einen konformen JWT-Token enthalten. Bevor dann die Logik ausgeführt wird, fährt jeder Request die einzelnen Komponenten der Middleware-Pipeline von .NET Core ab.

Die Authentifizierung und Autorisierung von Anfragen wird von der Middleware übernommen (siehe Abschnitt 2.7.6). Bei jeder Anfrage an die API wird vom Ressource Server des OAuth2-Service der Claim „NameIdentifier“ angefragt. Ist dieser vorhanden, kann mit der Durchführung der Geschäftslogik begonnen werden.

```
1     chat.UserGUID = (Guid)UserHelper.UserHelper.GetUserId(this);
```

Um für den Entwickler der Applikation ein nützliches Debugging bereitzustellen, werden in der Applikation benutzerdefinierte Exceptions im **Elasticsearch-Prozess** sowie im **OpenAI-API-Prozess** geworfen. Bei den Endpunkten werden diese Exceptions aufgefangen und passende Status-Codes zurückgegeben.

Die Businesslogik verwendet für die Verarbeitung von Daten Klassenbeschreibungen aus dem **Paket Panda.Domain**. Für den **Datentransfer** nach außen werden eigene **DTOs** verwendet. So erhalten die Entitäten ihre Unabhängigkeit und bleiben, für eine mögliche Weiterentwicklung des Backends, modifizierbar.

4.2.2 Frontendverwaltung

Das Frontend ist im Chat-Design aufgebaut. Jeder Benutzer speichert seine eigenen Chats, in denen er Anfragen an das Backend stellen kann. Die REST-API besitzt Endpunkte, um **CRUD-Operationen** auf das Chat- und ein LLM_Content-Objekt durchführen zu können.

Die mitgesendeten Objekte sind **DTOs** und erhalten alle eine Objektbeschreibung der Panda.Domain Klassenbibliothek mit dem **Automapper**-Paket. Die gemappten Objekte werden benötigt, um mit der Businesslogik zu kommunizieren. Für die Frontendverwaltung ist der ChatService zuständig.

Jeder Endpunkt der Frontendverwaltung benötigt die Zuteilung zu einem Benutzer. Dafür gibt es die **UserGuid**, welche aber nicht als Parameter übergeben wird. Sie ist im Zuge der Middleware-Pipeline (siehe Abschnitt 2.7.6) schon im Hauptspeicher abgelegt und muss nur noch über eine Hilfsmethode ausgelesen werden.

```
1  public static class UserHelper
2  {
3      public static Guid? GetUserId(this ControllerBase controller)
4      {
5          var idClaim = controller.User.FindFirst(ClaimTypes.NameIdentifier);
6          if (idClaim == null)
7          {
8              throw new InvalidOperationException("Null instance not set to an
                   object");
9          }
10         var userIdStr = idClaim.Value;
11
12         if (Guid.TryParse(userIdStr, out Guid userId))
13         {
14             return userId;
15         }
16
17         return null;
18     }
19 }
```

Abbildung 29: Hilfsmethode zum Auslesen des NameIdentifier-Claims.

In Zeile 5 wird über die ControllerBase der Claim mit dem Typen **ClaimTypes.NameIdentifier** gesucht und zurückgegeben. Wenn, aufgrund einer erfolgreichen Autorisierung, ein Benutzer anfragen darf, ist dieser Claim angelegt. In Zeile 10 und 12 wird der Wert des Claims ausgelesen und in Zeile 14 zurückgegeben.

Um im Frageantwortprozess noch bessere Ergebnisse zurückzubekommen, kann von außen noch ein Kontext mitgegeben werden, welcher als **Finetuning** in den Prozess eingearbeitet wird.

LLM_Content nennen wir dieses Objekt, das für die Chatanfragen einen Hilfstext repräsentiert. Signifikante Wörter einer Benutzerfrage werden von der OpenAI-API registriert und zurückgegeben. Diese API ist ein Large Language Model, das aufgrund eines Eingangstexts spezifische Antworten gibt. Das Objekt **LLM_Content** ist dafür zuständig, diesem Eingangstext von außen weitere Anweisungen hinzuzufügen (Finetuning), die helfen können den **Kontext** der Frage **besser zu verstehen**. Die hinzugefügten LLM_Contents müssen im Vorhinein in der Datenbank hinzugefügt werden und können im Frontend in einer Dropdown-Liste ausgewählt werden. Es folgen zwei Beispiele:

- Verwende aus der Frage Fachwörter und erstelle davon Synonyme, welche auch als Objekt an die Datenbank mitgegeben wird.
- Versuche, die wichtigsten Wörter aus der Frage in Englisch und in Deutsch herauszuschreiben.

4.2.3 Abfragen durchführen

Die **Benutzerfragen** werden im Frontend immer von einem Chat aus ausgeführt und die versendeten Objekte werden „**Query**“ genannt. Jede Query kann einen LLM_Content enthalten. Der Benutzer muss außerdem auswählen, ob die Suche in Elasticsearch oder Milvus stattfinden soll. Für die Abfrage einer Query gibt es zwei Endpunkte im Backend, welche Methoden des AnsweringService für die Beantwortung verwenden:

- PostQuery(QueryDTO) - übergibt eine Query und startet somit einen Frageantwortprozess (siehe Abschnitt 4.5.4). Die Query wird gemappt und im Beantwortungsprozess bearbeitet. Eine Antwort wird erstellt sowie Meta-Daten zugewiesen. Die Meta-Daten enthalten den Dateinamen, Autor, Dateityp und weitere Informationen des Dokuments, aus welchem die Antwort generiert wurde.
- UpdateQuery(QueryDTO) - aktualisiert das Query-Objekt. Wird hauptsächlich dafür verwendet, um die Gesamtdauer der Abfrage beim Objekt zu speichern. Dies ist für die Analyse im Nachhinein essenziell.

In der folgenden Abbildung ist der Endpunkte PostQuery zu sehen:

```
1  [HttpPost(query")]
2  [Consumes("application/json")]
3  public async Task<ActionResult<Task<QueryDTO>>> PostQuery([FromBody] QueryDTO
   ↪  queryDTO)
4  {
5      try
6      {
7          var (query, userGuid) = GetAuthenticatedQuery(queryDTO);
8          query = await ExecuteQuestionAsync(query);
9          await _chatService.AddQueryAsync(query, userGuid);
10         return Ok(_mapper.Map<QueryDTO>(query));
11     }
12     catch (InvalidOperationException ex)
13     {
14         if (ex.Message.Equals("Object not found")) return NotFound();
15         if (ex.Message.Equals("Null instance not set to an object")) return
           ↪  Unauthorized();
16         if (ex.Message.Equals("No chat found")) return NotFound("No chat found");
17         return BadRequest();
18     }
19     catch (Exception ex) when (ex is OpenAINotFoundException || ex is
       ↪  ElasticsearchNotFoundException)
20     {
21         return NotFound();
22     }
23     catch (Exception ex) when (ex is OpenAIConnectException || ex is
       ↪  OpenAIBadRequestException || ex is ElasticsearchConnectException)
24     {
25         return BadRequest();
26     }
27     catch (ArgumentOutOfRangeException ex)
28     {
29         return NotFound();
30     }
31 }
```

Abbildung 30: PostQuery-Endpunkt im DocumentsController

In Zeile 7 wird die Anfrage autorisiert und das DTO in ein Query-Objekt gemappt. In Zeile 8 wird die Query an die Methode des AnsweringService übergeben, welche die Beantwortung übernimmt. In Zeile 9 wird die nun vollständige Query in der Datenbank zur Archivierung gespeichert. Nach der Beantwortung und Speicherung befindet sich im Query die Antwort sowie die Meta-Daten des Dokuments. Diese werden als Response zurückgesendet.

4.3 Datenbanken

4.3.1 Elasticsearch

Implementierte Abfragearten

Im Backend wurden drei verschiedene Arten von Elasticsearch-Queries implementiert und getestet. Außerdem wurden zwei verschiedene Arten der Query-Generierung probiert. Gerade aufgrund der vielen Felder, die in den Dokumenten zur Verfügung gestanden sind, wäre die Anzahl der möglichen Queries um ein Vielfaches zu hoch, um sie zu testen und miteinander zu vergleichen. Folgende drei Queries wurden benutzt:

1. Der **Bool-Query** beschreibt, dass mindestens eins der folgenden Felder mit den übergebenen Werten übereinstimmen soll. Der Bool-Query ist in den Analysen der „default Query“, da er der erste war, welcher implementiert wurde. Bei diesem Query wurde auch die erste Strategie zur Generierung angewandt, bei der dieser nur als Struktur galt und komplett an das LLM gesendet wurde, in der Hoffnung, dass das LLM den Query anforderungsgemäß anpassen würde und die Platzhalter „1001“ mit Schlüsselwörtern befüllt. Da das LLM die Grundstruktur aber immer gleich gelassen hatte und teilweise auch Probleme hatte, die Platzhalter zu ersetzen, wurde bei den anderen beiden Queries eine andere Art der Query-Generierung genutzt.

```
{
    "query": {
        "bool": {
            "should": [
                {
                    "match": {
                        "meta.title": "1001"
                    }
                },
                {
                    "match": {
                        "content": "1001"
                    }
                },
                {
                    "match": {
                        "meta.fileName": "1001"
```

```
                    }
                }
            ],
            "minimum_should_match": 1
        }
    }
}
```

2. Bei diesem **Multi-Match-Query** wurde wie auch beim Nächsten eine andere Art der Query-Generierung verwendet. Bei dieser Variante wird nicht mehr der gesamte Query an das LLM gesendet und verarbeitet, sondern werden vom LLM nur die Schlüsselwörter aus der jeweiligen Frage extrahiert, welche dann im Backend selbst in die Platzhalter eingefügt wurden. Beim Multi-Match-Query wird auch wieder auf mehrere Felder gleichzeitig abgefragt. Auf alle drei Felder wird mit den gleichen Keywords gesucht. Die Notation, wie sie unten zu sehen ist, bedeutet, dass der Content 3-mal so viel und der Titel 2-mal so viel gewichtet ist wie der Autor. Der Typ beschreibt außerdem, wie die Relevanz bewertet werden soll. In diesem Fall wird das Feld mit der größten Übereinstimmung am meisten gewertet.

```
{
    "query": {
        "multi_match": {
            "query": "{escapedKeywords}",
            "fields": [
                "content^3",
                "meta.title^2",
                "meta.author"
            ],
            "type": "best_fields"
        }
    }
}
```

3. Der dritte und letzte Query ist eine **Kombination aus `must` und `should` Query**. Auch hier werden wieder nur die aus der Frage extrahierten Schlüsselwörter im Backend eingefügt. Bei diesem Query **muss** der gesuchte Begriff im Content vorkommen, kann aber auch im Feld Titel oder Autor vorkommen, wobei Titel doppelt so schwer gewichtet wird.

Der beim Content angegebene Operator „or“ ist relevant, wenn mehrere Schlüsselwörter angegeben werden, denn damit muss nur mindestens ein Wort übereinstimmen.

```
{
    "query": {
        "bool": {
            "must": [
                {
                    "match": {
                        "content": {
                            "query": "{escapedKeywords}",
                            "operator": "or"
                        }
                    }
                }
            ],
            "should": [
                {
                    "match_phrase": {
                        "meta.title": {
                            "query": "{escapedKeywords}",
                            "boost": 2
                        }
                    }
                },
                {
                    "match": {
                        "meta.author": {
                            "query": "{escapedKeywords}"
                        }
                    }
                }
            ]
        }
    }
}
```

Mehr zu den Generierungs- und Abfragearten findet sich im Kapitel Frage-Antwort-Prozess. Wie die verschiedenen Queryarten im Vergleich der Korrektheit der Antworten abgeschnitten haben, ist hier beschrieben: Datenbank Analysen.

4.3.2 Milvus

Installation von Milvus 3.5

Die **docker-compose.yml** Datei beschreibt, wie die verschiedenen Container aufgebaut und miteinander verbunden werden. Milvus benötigt zur Verwaltung und Speicherung von Daten mehrere unterstützende Dienste:

1. **etcd Service**

   Milvus nutzt etcd zur Verwaltung der Metadaten von Indexen und Vektorsätzen.

```yaml
etcd:
    container_name: milvus-etcd
    image: quay.io/coreos/etcd:v3.5.5
    environment:
      - ETCD_AUTO_COMPACTION_MODE=revision
      - ETCD_AUTO_COMPACTION_RETENTION=1000
      - ETCD_QUOTA_BACKEND_BYTES=4294967296
      - ETCD_SNAPSHOT_COUNT=50000
    volumes:
      - ${DOCKER_VOLUME_DIRECTORY:-.}/volumes/etcd:/etcd
    command: etcd -advertise-client-urls=http://127.0.0.1:2379
    ↪ -listen-client-urls http://0.0.0.0:2379 --data-dir /etcd
    healthcheck:
      test: ["CMD", "etcdctl", "endpoint", "health"]
      interval: 30s
      timeout: 20s
      retries: 3
```

   etcd ist eine verteilte Key-Value-Datenbank, die für verteilte Systeme entwickelt wurde. Sie dient zur Speicherung von Konfigurations- und Metadaten und wird von Milvus genutzt, um die Metadaten von Indexen und Vektorsätzen zu verwalten.[40]

2. **MinIO Service**

   MinIO ist eine skalierbare und S3-kompatible Objektspeicherlösung. Es wurde so entwickelt, dass es überall einsatzbereit ist - sei es in öffentlichen oder privaten Clouds, Bare-Metal-Servern oder in Edge-Infrastrukturen.

```yaml
minio:
    container_name: milvus-minio
    image: minio/minio:RELEASE.2023-03-20T20-16-18Z
    environment:
      MINIO_ACCESS_KEY: minioadmin
      MINIO_SECRET_KEY: minioadmin
    ports:
      - "9001:9001"
      - "9000:9000"
    volumes:
      - ${DOCKER_VOLUME_DIRECTORY:-.}/volumes/minio:/minio_data
    command: minio server /minio_data --console-address ":9001"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
      interval: 30s
      timeout: 20s
      retries: 3
```

   MinIO ermöglicht eine verteilte Speicherung und bietet eine schnelle Zugriffsmöglichkeit auf große Datenmengen. Milvus verwendet MinIO zur Verwaltung und Speicherung der Vektordaten, um eine schnelle Verarbeitung zu ermöglichen. [41]

3. **Milvus Standalone**

   Die Standalone-Version von Milvus fasst alle wichtigen Komponenten in einem einzigen Container zusammen. In Umgebungen mit höheren Anforderungen wird empfohlen, Milvus als Cluster mit mehreren Instanzen bereitzustellen.

```yaml
standalone:
    container_name: milvus-standalone
    image: milvusdb/milvus:v2.3.18
    command: ["milvus", "run", "standalone"]
    security_opt:
    - seccomp:unconfined
    environment:
```

```
      ETCD_ENDPOINTS: etcd:2379
      MINIO_ADDRESS: minio:9000
    volumes:
      - ${DOCKER_VOLUME_DIRECTORY:-.}/volumes/milvus:/var/lib/milvus
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9091/healthz"]
      interval: 30s
      start_period: 90s
      timeout: 20s
      retries: 3
    ports:
      - "19530:19530"
      - "9091:9091"
    depends_on:
      - "etcd"
      - "minio"
```

Die Konfiguration sorgt dafür, dass Milvus im Standalone-Modus läuft und mit etcd und MinIO verbunden ist. Milvus speichert seine Daten dauerhaft unter /var/lib/milvus. Ein Healthcheck überwacht, ob der Dienst zuverlässig läuft. Mit den Ports 19530 und 9091 ist es möglich, auf die Datenbank zuzugreifen.

Konfiguration der Milvus-Datenbank

Die Konfiguration der Datenbank erfolgt in der **appsettings.json** Datei. In der Datei sind die Verbindungsdetails und wichtige Einstellungen zu den Datenbanken hinterlegt. Durch diese Datei lassen sich die wichtigsten Einstellungen von Milvus in einer Datei ändern, ohne dass Änderungen im Code vorgenommen werden müssen.

```
"Milvus": {
  "Collection": "documents",
  "EmbedDimension": {
    "OpenAI": 1536,
    "Ollama": 384
  }
},
"AppConfig": {
  "Milvus": {
```

```
    "Limit": 3,
    "AnnsField": "vector"
  }
},
"ConnectionStrings": {
  "MilvusURI": "http://localhost:19530/"
}
```

Die Werte von der **appsettings.json** Datei, werden bei der Abfrage durch die Klasse **MilvusConfig** und bei der Speicherung durch **MilvusClient** geladen. Dadurch kann man Einstellungen an einer zentralen Stelle ändern, ohne überall im Code Anpassungen machen zu müssen. Das macht die Wartung einfacher und sorgt dafür, dass alles einheitlich bleibt. Das ist die Implementierung der MilvusConfig-Klasse:

```
public class MilvusConfig
{
    public string searchUri;
    public string collection;
    public int limit;
    public string annsField;

    public MilvusConfig(string uri, string collection, string annsField, int limit)
    {
        searchUri = uri;
        this.collection = collection;
        this.annsField = annsField;
        this.limit = limit;
    }
}
```

Speicherung von Daten in Milvus

Zum Speichern der Daten in die Datenbanken gibt es das Programm **DocumentDissector**, welches automatisch Dateien in einem Ordner einliest, in kleine Abschnitte (Chunks) unterteilt und anschließend in die Datenbank speichert. Durch die Verwendung von ein LLMs wie OpenAI oder der lokalen Version Ollama werden die Chunks in Vektoren umgewandelt, die für die Suche in der Vektordatenbank erforderlich sind. Die Speicherung der Daten in die Vektordatenbank wurde in dem Programm DocumentDissector in der Klasse **MilvusClient** implementiert.

Konstruktor MilvusClient Der Konstruktor der Klasse initialisiert die notwendigen Konfigurationswerte und erstellt eine HttpClient-Instanz. Er überprüft, mithilfe der **ManageCollection**, ob die Collection existiert, andernfalls wird eine neue erstellt. Außerdem wird überprüft ob die Anwendung lokal oder über OpenAI ausgeführt werden soll und setzt dementsprechend die Dimensionen der Vektoreinbettungen. Die Abfrage muss dann aber auch mit demselben LLM laufen, weil sonst die Dimensionen nicht korrekt sind.

```
1  public MilvusClient(IConfiguration configuration)
2   {
3       this.configuration = configuration;
4       if (configuration.GetValue<bool>("GlobalOptions:Run:Local"))
5       {
6           Dimension =
            ↪  configuration.GetValue<int>("GlobalOptions:Milvus:EmbedDimension:Ollama");
7       }
8       else
9       {
10          Dimension =
            ↪  configuration.GetValue<int>("GlobalOptions:Milvus:EmbedDimension:OpenAI");
11      }
12      Uri = configuration.GetValue<string>("ConnectionStrings:MilvusURI");
13
14      Client = new HttpClient();
15      // check if specified collection is given, otherwise create the collection
16      ManageCollection(configuration);
17  }
18
```

Collection-Verwaltung

Die Verwaltung der Collections in Milvus wird durch mehrere Methoden durchgeführt. Der Konstruktor der Klasse MilvusClient übernimmt die Initialisierung, während die Methoden ManageCollection(), GetCollectionExists() und CreateCollection() für die Verwaltung und Erstellung der Collections verantwortlich sind.

Die Methode **ManageCollection()** übernimmt die Organisation und Steuerung der Collections. Sie überprüft zunächst, ob eine bestimmte Collection bereits existiert. Falls nicht, wird eine

neue Collection mit den entsprechenden Parametern erstellt. Hier ist die Implementierung der Methode:

```
private async void ManageCollection(IConfiguration configuration)
 {
     MilvusCollectionManager manager = new MilvusCollectionManager(configuration);
     if (!await
      ↪  manager.GetCollectionExists(configuration.GetValue<string>("GlobalOptions:Milvus:Coll
         manager.CreateCollection();
     }
 }
```

Die Methode **GetCollectionExists()** überprüft per HTTP-GET-Anfrage an den Milvus-Server, ob eine bestimmte Collection existiert. Sie baut zunächst die URL für die Anfrage zusammen, sendet die Anfrage an den Milvus-Server und wertet die erhaltene Antwort aus.

```
public async Task<bool> GetCollectionExists(string collectionName)
{
    string requestUrl = _configuration.GetConnectionString("MilvusURI") +
                        "v1/vector/collections";
    Console.WriteLine(requestUrl);
    var response = await new HttpClient().GetAsync(requestUrl);
    string responseBody = await response.Content.ReadAsStringAsync();

    if (!response.IsSuccessStatusCode) throw new InvalidOperationException("No
     ↪  connection to milvus");

    var collectionsResponse =
     ↪  JsonConvert.DeserializeObject<CollectionListResponse>(responseBody);

    if (!collectionsResponse.Collections.Any()) return false;
    if (!collectionsResponse.Collections.Contains(collectionName)) return false;

    return true;
}
```

Funktionsweise:

- Baut die URL für die Abfrage zusammen.
- Führt eine **GET-Request** an Milvus aus.

- Falls die Collection existiert, gibt die Methode **true** zurück sonst **false**

Falls eine Collection nicht existiert, wird sie mit der Methode **CreateCollection()** erstellt. Dabei werden relevante Parameter wie Vektordimensionen und andere Einstellungen aus der appsettings.json geladen.

Diese Methode übernimmt folgende Aufgaben:

1. Lädt die Konfiguration aus der appsettings.json.
2. Erstellt eine neue Collection mit den definierten Vektor-Dimensionen und anderen Parametern.
3. Speichert die Collection in Milvus, sodass sie für künftige Anfragen bereitsteht.

```
1  public async void CreateCollection()
2  {
3      int vectorDimension = _configuration.GetValue<bool>("GlobalOptions:Run:Local")
4          ? _configuration.GetValue<int>("GlobalOptions:Milvus:EmbedDimension:Ollama")
5          :
           ↪  _configuration.GetValue<int>("GlobalOptions:Milvus:EmbedDimension:OpenAI");
6
7      string requestUrl = _configuration.GetConnectionString("MilvusURI") +
       ↪  "v2/vectordb/collections/create";
8      StringContent content = GetCreateRequestBody(vectorDimension,
       ↪  GetCustomFields(vectorDimension));
9
10     try
11     {
12         var response = await new HttpClient().PostAsync(requestUrl, content);
13         Console.WriteLine($"Status Code: {response.StatusCode}\nResponse: {await
           ↪  response.Content.ReadAsStringAsync()}");
14     }
15     catch (Exception ex)
16     {
17         throw new InvalidOperationException("Collection creation failed", ex);
18     }
19 }
```

Funktionsweise:

1. Bestimmt die Vektordimension basierend auf der Konfiguration

2. Baut den HTTP-Request-Body
3. Sendet einen **POST-Request** an Milvus
4. Gibt den Statuscode und die Antwort aus

Methode AddDocument

Als erstes wird das LLM ausgewählt:

```
ILLM llm;
if (configuration.GetValue<bool>("GlobalOptions:Run:Local"))
{
    llm = new OllamaLLM(...);
}
else
{
    llm = new OpenAILLM(...);
}
```

Anschließend wird der Dokumenteninhalt in einen Vektor umgewandelt:

```
float[] vectors = await llm.GetTextVectorAsync(document.Content);
```

Der generierte Vektor und das Dokument selbst werden in ein Dictionary gespeichert und an InsertDocIntoDatabase() übergeben:

```
Dictionary<string, object> doc = new Dictionary<string, object>
{
    { "vector", vectors },
    { "document", document }
};
await InsertDocIntoDatabase(doc);
```

Funktionsweise von InsertDocIntoDatabase()

Diese Methode ist eine asynchrone Methode, die ein Dictionary mit dem Vektor und dem Dokument als Parameter erhält und die Daten in Milvus einfügt.

1. **Erstellen des Request-URL**

```
string requestUri = Uri + "v1/vector/insert";
```

   - Die Basis-URL für Milvus wird aus den appsettings.json geladen (Uri).
   - Der Endpunkt für das Einfügen der Vektoren in Milvus ist v1/vector/insert

2. **Erstellen des Request-Bodys**

```
var requestBody = new
{
    collectionName =
    ↪  configuration.GetValue<string>("GlobalOptions:Milvus:Collection"),
    data = doc
};
```

   - Der Name der Collection wird aus appsettings.json geladen.
   - Die zu speichernden Daten (doc) enthalten den generierten Vektor und die Metadaten des Dokuments.

3. **Serialisierung in JSON**

```
var json = JsonConvert.SerializeObject(requestBody);
var content = new StringContent(json, Encoding.UTF8, "application/json");
```

   - Der requestBody wird in JSON konvertiert.
   - Ein StringContent-Objekt wird erstellt, um es als HTTP-Request-Body zu senden.

4. **Senden des POST-Requests an Milvus**

```
var response = await Client.PostAsync(requestUri, content);
var responseBody = await response.Content.ReadAsStringAsync();
```

   - Ein POST-Request mit dem JSON-Body wird an den Milvus-Server gesendet.
   - Die Antwort (responseBody) wird als string gespeichert.

5. **Überprüfung der Antwort**

```
if (!response.IsSuccessStatusCode) throw new InvalidOperationException("Insert
↪  not working");
```

   Falls der HTTP-Statuscode nicht erfolgreich ist, wird eine InvalidOperationException geworfen.

Ablauf der Vektorsuche

Die Vektorsuche besteht aus mehreren Schritten:

1. Generierung des Suchvektors aus der Benutzereingabe.
2. Senden des Vektors an Milvus und Vektorsuche, um ähnliche Dokumente zu finden.
3. Natürliche Antwort des Ergebnis bilden mit LLM

Generierung von Vektoren

Um eine semantische Suche durchführen zu können, müssen Texte in mathematische Vektoren umgewandelt werden. Dies geschieht mit einem Large Language Model (LLM). Die Vektorrechnung wird standardmäßig von OpenAI durchgeführt, weil die Lokale Variante mit Ollama nicht so gute Ergebnisse geliefert hatte. Die Auswahl des entsprechenden LLM kann in den **appsettings.json** konfiguriert werden und wird dynamisch in der **AnsweringService** Datei verwendet:

```
ILLM llm;
if (configuration.GetValue<bool>("GlobalOptions:Run:Local"))
{
    llm = new OllamaLLM(new OllamaConfig(...));
}
else
{
    llm = new OpenAILLM(new OpenAIConfig(...));
}
```

Nachdem das passende LLM ausgewählt wurde, wird der eingegebene Text in der Methode **ExecuteVectorAsync** in einen Vektor umgewandelt:

```
string vectorQuery = await
↪ llm.GetLLMResponseAsync(milvusConfig.GetOpenAIBuildQuery(query.Question,query.LLM_Content)
float[] inputVector = await llm.GetTextVectorAsync(vectorQuery);
```

Dieser Vektor wird dann an die Milvus-Datenbank gesendet, um eine Ähnlichkeitssuche durchzuführen.

Die Methode **ExecuteVectorAsync** führt die Vektorsuche in Milvus durch. Sie besteht aus drei Hauptschritten (Generierung des Vektors, Senden des Vektors an Milvus, Optimierung der Antwort). Von jedem Schritt wird die Zeit gemessen für die Analysen zwischen Milvus und Elastic:

```
async Task<Document> ExecuteVectorAsync(ILLM llm, Query query)
{
    Stopwatch stopwatch = Stopwatch.StartNew();
    Stopwatch stopwatchTotal = Stopwatch.StartNew();

    string vectorQuery = await
    ↪ llm.GetLLMResponseAsync(milvusConfig.GetOpenAIBuildQuery(query.Question,query.LLM_Cont
```

```
7       float[] inputVector = await llm.GetTextVectorAsync(vectorQuery);
8
9       stopwatch.Stop();
10      Console.WriteLine($"Milvus generate time: \t {stopwatch.ElapsedMilliseconds}");
11      stopwatch.Reset();
12      stopwatch.Start();
13
14      Document doc = await GetMilvusResponseAsync(inputVector);
15
16      stopwatch.Stop();
17      Console.WriteLine($"Milvus search time: \t {stopwatch.ElapsedMilliseconds}");
18      stopwatch.Reset();
19      stopwatch.Start();
20
21      doc.Content = await
        ↪  llm.GetLLMResponseAsync(milvusConfig.GetOpenAINaturalQuery(query.Question,
        ↪  doc.Content));
22
23      stopwatch.Stop();
24      Console.WriteLine($"Milvus natural time: \t {stopwatch.ElapsedMilliseconds}");
25      stopwatchTotal.Stop();
26      Console.WriteLine($"Milvus Total-Time: \t {stopwatchTotal.ElapsedMilliseconds}
        ↪  \n");
27
28      return doc;
29  }
```

Vektorsuche in Milvus

Der generierte Vektor wird mit der Methode **GetMilvusResponseAsync()** an Milvus gesendet, um eine Ähnlichkeitssuche durchzuführen. Die Methode sendet eine HTTP-Anfrage an die Milvus-Datenbank, diese verarbeitet die Antwort und gibt das entsprechende Dokument zurück, das der Suchanfrage am nächsten kommt. Hier der Aufruf der Methode:

```
1  Document doc = await GetMilvusResponseAsync(inputVector);
```

Die aufgerufene Methode **GetMilvusResponseAsync()** ist asynchron und erwartet einen Eingabevektor als Parameter. Sie kümmert sich um die Kommunikation mit Milvus und

behandelt verschiedene Fehlerszenarien, die während des Prozesses auftreten können. Hier die Implementierung der Methode:

```
public async Task<Document> GetMilvusResponseAsync(float[] inputVector)
{
    var requestUri = milvusConfig.searchUri;
    var requestBody = new
    {
        collectionName = milvusConfig.collection,
        data = new[]
        {
        inputVector
    },
        milvusConfig.annsField,
        milvusConfig.limit,
        outputFields = new[] { "id", "vector", "document" }
    };

    var json = JsonConvert.SerializeObject(requestBody);
    var content = new StringContent(json, Encoding.UTF8, "application/json");

    HttpResponseMessage response;
    string responseBody;
    try
    {
        response = await new HttpClient().PostAsync(requestUri, content);
        responseBody = await response.Content.ReadAsStringAsync();
    }
    catch (HttpRequestException)
    {
        throw new InvalidOperationException("No connection to milvus");
    }
    catch (Exception)
    {
        throw new InvalidOperationException("Milvus: Mismatch of Dimensions");
    }

    dynamic result = JsonConvert.DeserializeObject(responseBody)!;
```

```
36      if (result.code == 100) throw new InvalidOperationException("No collection
        ↪  found");
37      if (result.data == null) throw new InvalidOperationException("Milvus: Bad
        ↪  Request");
38
39      //handling the dynamic type with strong json type
40      JObject firstDataObject = result.data[0].document;
41      return firstDataObject.ToObject<Document>()!;
42  }
```

1. **Erstellung der Anfrage**

   Zunächst wird in Zeile 3 die URL für Milvus aus der Konfigurationsdatei geladen. Anschließend wird ein JSON-Object erstellt, das die notwendigen Parameter für Suche enthält:

   - **collectionName:** Name der Collection, in der gesucht werden soll.
   - **data:** Der Eingabevektor, der mit den gespeicherten Vektoren verglichen wird.
   - **annsField:** Das Attribut, welches für die Suche verwendet werden soll in unserem Fall ist das **Vektor**
   - **limit:** Die Anzahl der zurückzugebenen ähnlichen Ergebnisse.
   - **outputFields:** Die Felder, die in der Antwort enthalten sein sollen (id, vector, document)

   Das erstellte JSON-Objekt wird mit JsonConvert.SerializeObject(requestBody) in eine Zeichenkette umgewandelt. Danach wird es in eine StringContent-Instanz verpackt, um es als HTTP-Request-Body verwenden zu können.

2. **Senden der HTTP Anfragen**

   Die Anfrage wird mit **PostAsync()** in Zeile 23 an Milvus gesendet. Da es im um eine asynchrone Methode handelt, blockiert es den Hauptthread nicht.

   - **PostAsync()** sendet die Anfrage an Milvus
   - **ReadAsStringAsync()** liest die Antwort des Servers als Zeichenkette aus.

3. **Verarbeitung der Antwort**

   Nach erfolgreicher Kommunikation mit Milvus wird die Antwort in ein **dynamic**-Objekt deserialisiert:

```
1 dynamic result = JsonConvert.DeserializeObject(responseBody)!;
```

Anschließend werden verschieden Überprüfungen durchgeführt:

- **result.code == 100**, die Collection konnte nicht gefunden werden
- **result.code == null**, bedeutet das die Anfrage fehlerhaft ist

Wenn die Antwort gültig ist, wird das Dokument aus **result.data[0].document** extrahiert und in ein **Document** umgewandelt.

LLM-generierte natürliche Antwort aus Suchergebnis

Nachdem Milvus das beste Dokument zur Suchanfrage zurückgegeben hat, wird das beste gefundene Dokument weiterverarbeitet. Die Weiterverarbeitung geschieht mit einem Large Language Model, das auf Grundlage des Ergebnisses die Frage beantwortet. Die Methode wird in der Methode **ExecuteVectorAsync()** in Zeile 21 aufgerufen:

```
1     doc.Content = await
      ↪ llm.GetLLMResponseAsync(milvusConfig.GetOpenAINaturalQuery(query.Question,
      ↪ doc.Content));
```

Die Methode **GetOpenAINaturalQuery()** formatiert die ursprüngliche Benutzerfrage zusammen mit dem Ergebnis zu einer strukturierten Eingabe.

```
1  public string GetOpenAINaturalQuery(string question,string content)
2  {
3      string todo = $@"Verwende folgende Information um die Frage am Schluss in
       ↪ natürlicher Sprache zu beantworten. Erfinde selbst nichts dazu.
4
5                      Information: ${content}
6
7                      Frage: ${question}
8                      Hilfreiche Antwort";
9      return todo;
10 }
```

Durch diesen Schritt wird das Suchergebnis der Datenbank nicht nur zurückgegeben, sondern in eine verständliche Antwort umgewandelt, die direkt die gestellte Frage beantwortet.

4.3.3 Frontend-Datenbank

Im Projekt nutzen wir eine MS-SQL-Datenbank zur Speicherung der Chats der Benutzer. Das zugehörige Datenmodell, das mit Entity Framework Core implementiert wurde (siehe Abbildung 31), basiert auf einem relationalen Modell. Es ermöglicht uns eine effiziente Verwaltung, Speicherung und Abfrage der Daten.

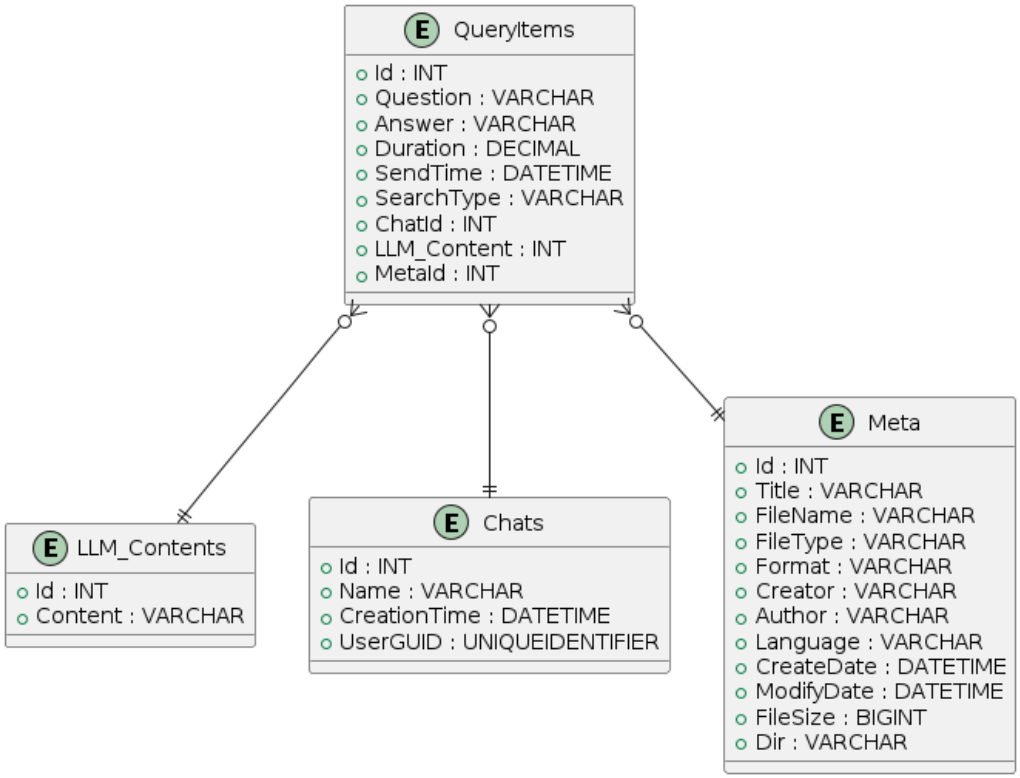


Abbildung 31: Datenmodell-Frontend-DB

Meta:

Die Tabelle Meta speichert die Metadaten eines Dokuments. Sie enthält den Primärschlüssel „Id“, der jedes Dokument eindeutig identifiziert. Die restlichen Attribute, die im Datenmodell (siehe Abbildung 31) dargestellt sind, speichern verschiedene Metadaten, die ein Dokument haben kann. Es bleiben viele Spalten oft leer, da bei den meisten Dokumenten nicht alle Metadaten bekannt sind.

Chats:

Die Tabelle Chats speichert alle Chats und die dazugehörigen Informationen.

Die Tabelle hat folgende Attribute:

Name des Attributs	Datentyp	Beschreibung
Id	INT	Die „Id“ ist der Primärschlüssel dieser Tabelle und identifiziert jeden Chat.
Name	VARCHAR	Speichert den Namen des Chats.
CreationTime	DATETIME	Speichert den genauen Zeitpunkt der Erstellung des Chats.
UserGUID	UNIQUE IDENTIFIER	Dieses Attribut speichert die GUID des Benutzers, der den Chat angelegt hat. Die Benutzer loggen sich über das SSO von ITPRO ein und von dort erhält das System die GUID des eingeloggten Benutzers.

LLM_Contents:

Die Tabelle LLM_Content speichert die verschiedenen LLM-Contents, die der Benutzer auswählen kann. Das Backend verfügt über CRUD-Methoden und REST-Endpunkte für den Content, im Frontend selbst gibt es aber keine Möglichkeiten, um einen LLM-Content in die Tabelle hinzuzufügen oder zu löschen. Alle vorhandenen LLM-Contents müssen manuell über die Endpunkte in die Datenbank eingefügt werden.

Die Tabelle LLM_Contents hat folgende Attribute:

Name des Attributs	Datentyp	Beschreibung
Id	INT	Die „Id“ ist der Primärschlüssel dieser Tabelle und identifiziert jeden LLM-Content.
Content	VARCHAR	Speichert den eigentlichen Inhalt des LLM-Contents.

QueryItem:

Die Tabelle QueryItem speichert die Benutzerfragen und die dazugehörigen Informationen.

Die Entität besteht aus folgenden Attributen:

Name des Attributs	**Datentyp**	**Beschreibung**
Id	INT	Die „Id“ ist der Primärschlüssel dieser Tabelle und identifiziert jeden QueryItem eindeutig.
Question	VARCHAR	Dieses Attribut speichert die Frage, die der Benutzer gestellt hat.
Answer	VARCHAR	Dies ist die Antwort, die das System auf die Frage generiert hat.
Duration	DECIMAL	Die Zeit in Millisekunden, die benötigt wurde, um die Antwort vom Backend zu erhalten.
SendTime	DATETIME	Speichert den Zeitpunkt, zu dem das QueryItem erstellt wurde.
SearchType	VARCHAR	Speichert, mit welcher Suche die Antwort ermittelt wurde. Es gibt nur zwei mögliche Werte: „Lexical“ für die lexikalische Suche und „Vector“ für die Vektorsuche.
ChatId	INT	Die „ChatId“ ist ein Fremdschlüssel zur Tabelle Chats und bestimmt daher, zu welchem Chat das QueryItem gehört.
LLM_Content	INT	Fremdschlüssel zur Tabelle LLM_Contents. Gibt an, welcher LLM_Content zur Beantwortung der Frage verwendet wurde.
MetaId	INT	„MetaId“ ist ein Fremdschlüssel zur Tabelle Meta und speichert daher, in welchem Dokument die Antwort auf die gestellte Frage gefunden wurde.

4.4 Webanwendung

4.4.1 Einleitung

Über die Webanwendung kann der Benutzer auf das System zugreifen. Nach der erfolgreichen Anmeldung über das SSO hat der Benutzer die Möglichkeit, eine Frage zu stellen. Das System verarbeitet die Frage und liefert eine Antwort zurück, die dem Benutzer angezeigt wird.

Zur Umsetzung der Webanwendung wurde das Web-Framework Angular verwendet.

4.4.2 Aufbau

Die Webanwendung ist eine Angularanwendung, die aus mehreren Komponenten besteht, welche verschiedene Funktionen haben. Sie ist durchgehend mit dem SSO der ITPRO verbunden, um den Zugriff auf die Anwendung nur für berechtigte Benutzer zu ermöglichen. Zusätzlich sorgt ein Route-Guard dafür, dass nicht authentifizierte Benutzer keinen Zugang zu geschützten Seiten haben. Durch den Silent-Refresh des Access-Tokens bleibt der Benutzer dauerhaft angemeldet. Alle zentralen Funktionen befinden sich auf einer Seite, die aus mehreren Komponenten besteht. Die Kommunikation zwischen der Webanwendung und dem Backend erfolgt über eine REST-API, die in Services eingebunden ist. Damit mehrere Komponenten und Services auf die gleichen Daten zugreifen können, wird NGXS für das State Management verwendet.

4.4.3 Funktionen

Anmeldung

Falls der Benutzer keinen gültigen Access-Token besitzt, wird er automatisch zur LoginPage-Component weitergeleitet. In der Komponente wird sofort die login-Methode des AuthService aufgerufen, welche den Benutzer direkt auf die SSO-Login-Seite (siehe Abbildung 32) von ITPRO weiterleitet.

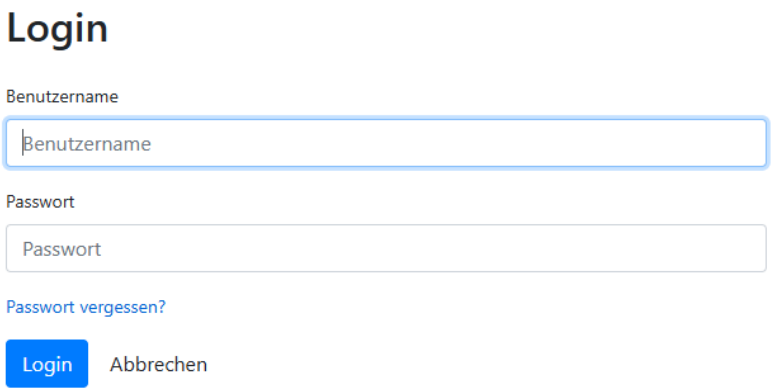


Abbildung 32: SSO Login-Seite

Bei gültiger Anmeldung wird der Benutzer automatisch auf die HomePageComponent weitergeleitet, wo er alle seine Chats und Abfragen sieht. Außerdem hat der Benutzer nach der Anmeldung auch Zugriff auf alle Anwendungen, die zur Authentifizierung dasselbe SSO von ITPRO verwenden.

Abmeldung

Abbildung 33: „Abmelde“-Button Frontend

Um sich abzumelden, muss der Benutzer auf den „Abmelde“-Button (siehe Abbildung 33) klicken. Beim Klick auf den Button wird die logout-Methode vom AuthService aufgerufen. Diese macht die Token ungültig und meldet den Benutzer vom SSO ab.

Da der Routing-Guard nicht authentifizierte Benutzer automatisch zur LoginPageComponen weiterleitet, beginnt der Anmeldeprozess erneut und der Benutzer kommt wieder zur SSO-Anmeldeseite.

Anzeige der Chat-Liste

Abbildung 34: Chatliste Frontend

Die Chatliste (siehe Abbildung 34) zeigt alle Chats des eingeloggten Benutzers an.

- Beim Klick auf das Menü-Icon oben links in der Ecke kann die Chatliste ein- oder ausgeblendet werden.
- Mit einem Klick auf das eingekreiste Plus-Icon oben rechts in der Ecke wird ein neuer Chat zur Liste hinzugefügt.
- Bei Doppelklick auf den Namen des Chats kann der Name bearbeitet werden.
- Bei einem Klick auf das Mülleimer-Icon rechts neben dem Namen wird der Chat gelöscht.

Stellen einer Frage

Beim Stellen einer Frage können mehrere Konfigurationen vorgenommen werden, die die Antwort verändern können.

- Mit dem Toggle (siehe Abbildung 35) oben rechts im Fenster kann zwischen lexikalischer Suche und Vektorsuche umgeschaltet werden. Es ist für die lexikalische Suche und bedeutet Elastic, da die lexikalische Suche durch Elasticsearch erfolgt und das M bedeutet Milvius, da die Vektorsuche mit Milvius erfolgt.

Abbildung 35: Suchoptionen-Toggle Frontend

- Weiters gibt es noch zwei Input-Felder (siehe Abbildung 36):
  - Im oberen Input-Feld kann der gewünschte LLM-Input ausgewählt werden, der einen großen Einfluss auf die Antwort der Frage hat. Falls der Benutzer keinen LLM-Content auswählt, wird ein Standard-LLM-Content verwendet.
  - Im unteren Input-Feld wird die Frage eingegeben.

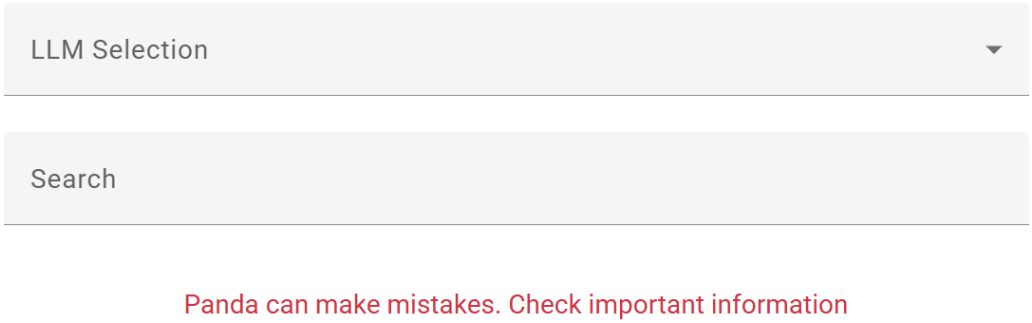


Abbildung 36: Input-Felder Frontend

Nachdem die Frage abgeschickt wurde und das Backend eine Antwort zurückgeliefert hat, werden die Frage und die Antwort angezeigt. Zusätzlich zur Antwort wird auch der Autor des Dokuments, in dem die Antwort gefunden wurde, sowie die benötigte Zeit zur Beantwortung und die Art der Suche, angezeigt.

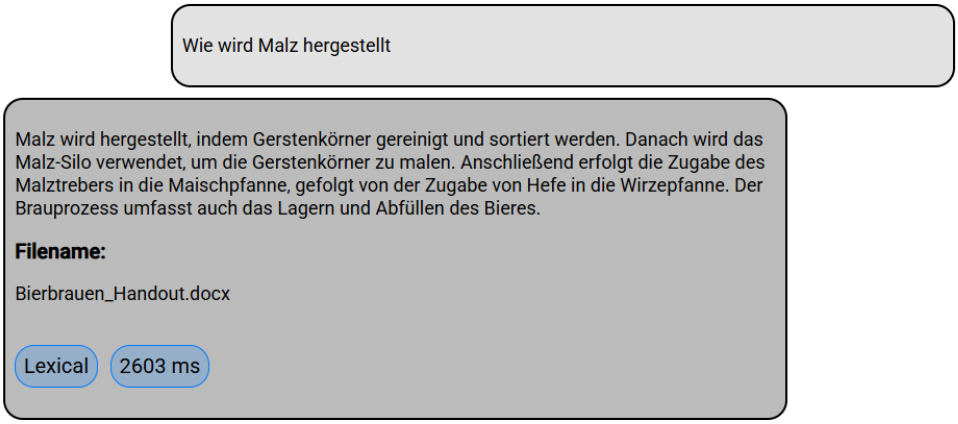


Abbildung 37: Ausgabe Frontend

4.4.4 State Management

In unserer Webanwendung verwenden wir NGXS für das State Management, um den Zustand der Applikation zentral zu speichern und zu verwalten. Eine genauere Erklärung zu State Management mit NGXS ist im Kapitel NGXS zu finden.

Das AppStateModel definiert die Struktur der Daten, die im State gespeichert werden.
Hier folgt eine genauere Erklärung über das AppStateModel.

```
1 export interface AppStateModel {
2   httpHeader: HttpHeaders;
3   currentUser?: User;
4   userChats?: Chat[];
```

```
5   currentChat?: Chat;
6   currentDb?: string;
7   chatQueries?: Query[];
8   llmOptions?: Llm[];
9 }
```

httpHeader speichert den HTTP-Header, der bei jeder API-Anfrage an das Backend mitgesendet wird.

currentUser enthält den aktuell angemeldeten Benutzer.

userChats speichert eine Liste von allen Chats des eingeloggten Benutzers.

currentChat enthält den aktuell ausgewählten Chat.

currentDb speichert die aktuell ausgewählte Datenbank, die für die Suche nach der Antwort verwendet wird.

chatQueries speichert eine Liste von allen Queries, die zum ausgewählten Chat gehören.

llmOptions speichert eine Liste von allen LLM-Optionen, die der Benutzer auswählen kann.

In der AppState-Klasse sind mehrere Selektoren implementiert, die es ermöglichen, auf die Daten zuzugreifen. Die Selektoren werden in den Komponenten und Services verwendet, um den aktuellen Zustand der Daten abzurufen.

```
1  @Selector()
2    public static httpHeader(state: AppStateModel): HttpHeaders{
3      return state.httpHeader;
4    }
5    @Selector()
6    public static currentUser(state: AppStateModel): User | null {
7      return state.currentUser ?? null;
8    }
9    @Selector()
10   public static userChats(state: AppStateModel): Chat[]{
11     return state.userChats || [];
12   }
13   @Selector()
14   public static currentChat(state: AppStateModel): Chat | null {
15     if (!state.currentChat) { return null; }
16     return { ...state.currentChat } as Chat;
17   }
```

```
18    @Selector()
19    public static currentDb(state: AppStateModel): string | null {
20      return state.currentDb ?? null;
21    }
22    @Selector()
23    public static chatQueries(state: AppStateModel): Query[]{
24      return state.chatQueries || [];
25    }
26    @Selector()
27    public static llmOptions(state: AppStateModel): Llm[]{
28      return state.llmOptions || [];
29    }
```

Die Selektoren ermöglichen den Zugriff auf alle im AppState gespeicherten Attribute. Sie geben den aktuellen Wert des jeweiligen Attributs zurück und falls ein Wert nicht definiert ist, wird ein Standardwert zurückgegeben.

Aktionen werden verwendet, um den Zustand der Anwendung zu ändern. Jede Action beschreibt eine bestimmte Änderung im State. Die Action in unserer Anwendung werden in separate Action-Klassen definiert und in der AppState-Klasse selbst wird die Logik implementiert, die bestimmt, wie der State durch die Action verändert wird.
Die Implementierung der Actions in der AppState-Klasse sieht wie folgt aus:

```
1  @Action(SetUser)
2    public setCurrentUser({ setState }: StateContext<AppStateModel>, { payload }:
     ↪  SetUser) {
3      setState((state: AppStateModel) => ({
4        ...state,
5        currentUser: payload
6      }));
7    }
```

Die SetUser-Action setzt einfach das currentUser-Attribut im State auf den Wert des Payload-Objektes.

Die FetchUserChats-Action dient dazu, die Chats vom eingeloggten Benutzers vom Backend abzurufen und diese dann im userChats-Attribut zu speichern.

```
@Action(FetchUserChats)
  async fetchUserChats(ctx: StateContext<AppStateModel>){
    try{
      const chats = await this.chatService.getUserChats().toPromise();
      ctx.dispatch(new SetUserChats(chats));
    }catch (error){
      console.error("Error fetching user chats: ",error);
    }
  }
```

Zuerst werden mithilfe des ChatService die Chats vom eingeloggten Benutzer vom Backend geladen. Anschließend wird die SetUserChats-Action aufgerufen, um die abgerufenen Chats im userChats-Attribut zu speichern.

Die AddChat-Action wird verwendet, um einen neuen Chat zur userChats-Liste hinzuzufügen.

```
@Action(AddChat)
  async addChat(ctx: StateContext<AppStateModel>) {
    try{
      const newChat = await this.chatService.addNewChat().toPromise();
      if(newChat){
        const state = ctx.getState();
        const userChats: any = state.userChats ? [newChat, ...state.userChats] :
        ↪  [newChat];
        ctx.patchState({ userChats });
        ctx.dispatch(new SetChat(newChat as Chat));
      }else {
        console.error("Received chat is undefined")
      }
    }catch (error){
      console.error("Error adding new chat: ",error)
    }
}
```

Die AddChat-Action ruft zuerst die addNewChat-Methode vom ChatService auf, um einen neuen Chat im Backend zu erstellen. Falls die Erstellung des Chats erfolgreich ist, wird die aktuelle userChats-Liste aus dem State geholt und um den neu erstellten Chat erweitert. Anschließend

wird das userChats-Array aktualisiert und currentChat wird auf den neu erstellten Chat gesetzt.

Die DeleteChat-Action wird verwendet, um einen Chat aus dem Backend und dem State zu entfernen.

```
@Action(DeleteChat)
  async deleteChat(ctx: StateContext<AppStateModel>, {payload}: DeleteChat) {
    try{
      this.chatService.deleteChat(payload!);
      const state = ctx.getState();
      const userChats = state.userChats?.filter(chat => chat.id !== payload?.id);
      ctx.patchState({userChats});
      if (state.currentChat?.id === payload?.id) {
        ctx.dispatch(new SetFirstChatAsCurrent());
      }
    }catch (error){
      console.error("Error deleting chat: ",error);
    }
}
```

Als Payload erwartet die DeleteChat-Action das zu löschende Chat-Objekt. Um den Chat aus dem Backend zu löschen, wird im ChatService die deleteChat-Methode aufgerufen. Anschließend wird die userChats-Liste im State aktualisiert, indem der gelöschte Chat mithilfe der filter-Methode entfernt und anschließend wieder im State gespeichert wird, sodass die Änderungen wirksam sind. Zum Schluss wird die SetFirstChatAsCurrent-Action aufgerufen, um das currentChat-Attribut auf einen neuen Wert zu setzen, falls der gelöschte Chat currentChat war.

Die UpdateChat-Action wird verwendet, um einen Chat zu bearbeiten.

```
@Action(UpdateChat)
updateChat(ctx: StateContext<AppStateModel>, { payload }: UpdateChat) {
    this.chatService.updateChat(payload!).subscribe({
      next: () => {
        const state = ctx.getState();
        const updatedUserChats: any = state.userChats?.filter(chat => chat.id !==
        ↪  payload?.id);
        updatedUserChats.push(payload);
        ctx.dispatch(new SetUserChats(updatedUserChats));
```

```
      },
      error: (error) => {
        console.error("Error updating chat: ", error);
      }
    });
}
```

Als Payload erwartet die Action das Chat-Objekt, das bearbeitet werden soll. Zuerst wird die updateChat-Methode des ChatService aufgerufen, um den Chat im Backend zu ändern. Wenn der Request erfolgreich war, wird das alte Chat-Objekt von der Liste entfernt und das aktualisierte Chat-Objekt hinzugefügt. Anschließend wird der State mit der aktualisierten Liste der User-Chats geändert.

Die SetUserChats-Action dient dazu, die userChats-Liste im State zu aktualisieren und nach dem Erstellungszeitpunkt zu sortieren.

```
@Action(SetUserChats)
  public setUserChats({setState}: StateContext<AppStateModel>, { payload }:
  ↪ SetUserChats) {
    payload?.sort((a,b) => {
      const dateA = new Date(a.creationTime!).getTime();
      const dateB = new Date(b.creationTime!).getTime();
      return dateB < dateA ? -1 : 1;
    });
    setState((state: AppStateModel) => ({
      ... state,
      userChats: payload
    }));
  }
```

Die SetUserChats-Action erwartet als Payload die Chat-Liste, auf die userChats gesetzt werden soll. Zuerst wird diese Liste absteigend nach creationTime sortiert, damit der neuste Chat an der ersten Stelle steht. Danach wird das userChats-Attribut mit dem sortierten Payload aktualisiert.

Die SetChat-Action dient dazu, den ausgewählten Chat im State zu speichern.

```
@Action(SetChat)
  public setChat({setState, dispatch}: StateContext<AppStateModel>, { payload }:
  ↪ SetChat) {
    setState((state: AppStateModel) => ({
      ...state,
      currentChat: payload
    }));
    if (payload && payload.queryItems)
      dispatch(new SetQueries(payload.queryItems));
    else
      dispatch(new SetQueries([]))
}
```

Die Action erwartet als Payload das Chat-Objekt, das auserwählt werden soll. Nachdem das currentChat-Attribut mit dem Payload aktualisiert wurde, wird die SetQueries-Action aufgerufen, um die QueryItems des Chats separat zu speichern. Falls keine queryItems vorhanden sind oder currentChat auf 'undefined' gesetzt wurde, wird eine leere Liste übergeben.

Die ausgewählte Suchdatenbank wird mithilfe der SetDb-Action im State gespeichert.

```
@Action(SetDb)
  public setCurrentDb({setState}: StateContext<AppStateModel>, { payload }: SetDb) {
    setState((state: AppStateModel) => ({
      ...state,
      currentDb: payload
    }));
}
```

Als Payload erwartet die Action die Suchdatenbank, die genutzt werden soll. Sobald die Methode aufgerufen wird, wird das currentDb-Attribut im State mit dem Payload aktualisiert.

Die SetQueries-Action aktualisiert das chatQueries-Attribut im State.

```
@Action(SetQueries)
  public setChatQueries({setState}: StateContext<AppStateModel>, { payload }:
  ↪ SetQueries) {
    setState((state: AppStateModel) => ({
```

```
      ...state,
      chatQueries: payload
    }));
  }
```

Als Payload erwartet die Action das Query-Array, auf das chatQueries gesetzt werden soll.

Die AddQuery-Action wird verwendet um ein neues Query-Objekt im State hinzuzufügen.

```
@Action(AddQuery)
  async addQuery(ctx: StateContext<AppStateModel>, {payload}: AddQuery) {
    try{
      const newQuery = payload;
      if(newQuery){
        const state = ctx.getState();
        const currentChatId = state.currentChat?.id;
        const updatedUserChats = state.userChats?.map(chat => {
          if (chat.id === currentChatId) {
            return {
              ...chat,
              queryItems: chat.queryItems ? [...chat.queryItems, newQuery] :
              ↪ [newQuery]
            };
          }
          return chat;});
        if (updatedUserChats) {
          const updatedChat = updatedUserChats.find(chat => chat.id ===
          ↪ currentChatId);
          if (updatedChat) {
            ctx.dispatch(new SetUserChats(updatedUserChats))
            ctx.dispatch(new SetChat(updatedChat));
          }
        }
      }
    }catch (error){
      console.error("Error adding new Query: ",error)
    }
}
```

Als Payload erwartet die Action das Query-Objekt, das hinzugefügt werden soll. Damit der gesamte State auf dem richtigen Stand bleibt, muss das Query-Objekt sowohl in chatQueries, im currentChat und im richtigen Chat in der userChats-Liste hinzugefügt werden. Als erstes wird der Query im richtigen Chat im userChats-Array hinzugefügt und anschließend wird der geänderte Chat vom Array herausgefiltert. Wenn das alles erfolgreich war, wird zuerst die SetUserChats-Action aufgerufen, um userChats im State zu aktualisieren und danach die SetChat-Action, damit currentChat auch aktualisiert wird. In der SetChat-Action wird die SetQueries-Action aufgerufen, deshalb ist danach chatQueries auch auf dem richtigen Stand.

Durch die UpdateQuery-Action wird ein Query überall im Programm richtig geändert.

```
@Action(UpdateQuery)
async updateQuery(ctx: StateContext<AppStateModel>,{payload}: UpdateQuery){
    try{
      this.queryService.updateQuery(payload!).subscribe({
        next: (updatedQuery) => {
          const state = ctx.getState();
          const updatedUserChats = state.userChats?.map(chat =>
            chat.id === updatedQuery.chatId
              ? {
                ...chat,
                chatQueries: chat.queryItems.map(query =>
                  query.id === updatedQuery.id ? updatedQuery : query
                ),
              }
              : chat
          );
          if (updatedUserChats) {
            const updatedChat = updatedUserChats.find(chat => chat.id ===
            ↪  updatedQuery.chatId);
            if (updatedChat) {
              ctx.dispatch(new SetUserChats(updatedUserChats));
              ctx.dispatch(new SetChat(updatedChat));
            }}},
        error: (error) => {
          console.error("Error updating query: ", error);
        }
```

```
26        });
27      }catch (error){
28        console.error("Error updating query: ",error);
29      }
30  }
```

Die Action erwartet als Payload das geänderte Query-Objekt. Als Erstes wird die updateQuery-Methode im QueryService aufgerufen, um das Query im Backend zu aktualisieren. Wenn der Request erfolgreich war, wird das Query überall im State angepasst. Dafür wird zuerst in der userChats-Liste im richtigen Chat in dessen queryItems-Liste das alte Query mit dem aktualisierten Query ausgetauscht. Zum Schluss wird im State userChats auf die neue Liste gesetzt und die SetChat-Action aufgerufen, damit der Chat, in dem das aktualisierte Query ist, angezeigt wird und die chatQueries-Liste ebenfalls auf dem richtigen Stand bleibt.

```
1  @Action(SetLlmOptions)
2    async setLlmOptions({ setState }: StateContext<AppStateModel>, {payload}:
     ↪  SetLlmOptions){
3      try{
4        setState((state: AppStateModel) => ({
5          ...state,
6          llmOptions: payload
7        }));
8      }catch (error){
9        console.error("Error getting llm options", error);
10     }
11 }
```

Die SetLlmOptions-Action wird verwendet, um das llmOptions-Attribut auf den übergebenen Payload zu setzen. Der Payload ist ein Array von Llm-Objekten, das direkt in llmOptions gespeichert wird.

```
1  @Action(GetLlmOptions)
2    async getLlmOptons(ctx: StateContext<AppStateModel>){
3      const llms = await this.llmService.getLlms().toPromise();
4      ctx.dispatch(new SetLlmOptions(llms));
5  }
```

Die GetLlmOptions-Action ruft die getLlms-Methode im LlmService auf, um alle Llm-Optionen vom Backend zu holen. Anschließend wird die SetLlmOptions-Action aufgerufen, um die Llm-Optionen im State zu speichern.

```
@Action(SetFirstChatAsCurrent)
  public setFirstChatAsCurrent(ctx: StateContext<AppStateModel>){
    const state = ctx.getState();
    const firstChat = state.userChats && state.userChats.length > 0 ?
    ↪  state.userChats[0] : undefined;
    if (firstChat){
      ctx.dispatch(new SetChat(firstChat));
    }else {
      ctx.dispatch(new SetChat(undefined));
    }
}
```

Die SetFirstChatAsCurrent-Action setzt das currentChat-Attribut auf den ersten Chat der userChats-Liste, dadurch automatisch auch auf den neusten Chat. Dafür wird einfach die SetChat-Action aufgerufen und das erste Element der userChats-Liste mitgegeben und wenn kein Chat vorhanden ist wird, in der SetChat-Action 'undefined' als Payload mitgegeben.

4.4.5 Service

AuthService

AuthService stellt die Methoden zur Authentifizierung zur Verfügung. Der Service besteht aus zwei wichtigen Methoden: **login** und **logout**. Für die Implementierung wird die angular-oauth-oidc Bibliothek verwendet.

Die Login Methode führt die Anmeldung des Benutzers durch. In der Methode werden als erstes die Konfigurationen vorgenommen.

```
const authCodeFlowConfig: AuthConfig = {
      issuer: 'https://sso.itpro.at/identity',
      clientId: 'PANDA_FRONTEND',
      responseType: 'code',
      scope: 'openid profile IdentityServerApi PandaApi offline_access',
      silentRefreshRedirectUri: `${window.location.origin}/silent-refresh.html`,
      useSilentRefresh: true,
```

```
8       timeoutFactor: 0.75,
9       sessionChecksEnabled: true,
10      clearHashAfterLogin: true,
11      redirectUri: window.location.origin + this.locationStrategy.getBaseHref(),
12      postLogoutRedirectUri: window.location.origin +
        ↪  this.locationStrategy.getBaseHref(),
13      logoutUrl: window.location.origin + this.locationStrategy.getBaseHref(),
14      revocationEndpoint: 'https://sso.itpro.at/identity/connect/revocation'
15    };
```

issuer: Issuer gibt die URL des Identity Providers an.

clientId: ClientId gibt an, welcher Client sich Authentifizieren möchte. Die ClientId muss beim Authorization Server hinterlegt werden.

responseType: Der ResponseType gibt an was als Response erwartet wird. ResponseType: 'code' bedeutet, dass Code Flow verwendet wird.

scope: Die Rechte, die der Benutzer nach der Anmeldung hat.

silentRefreshRedirectUri: Gibt den Pfad an, der für das Silent Refresh verwendet wird.

useSilentRefresh: UseSilentRefresh gibt an, ob Silent Refresh für die automatische Token-Erneuerung verwendet werden soll. Da die Anwendung Silent-Refresh verwendet, ist der Wert auf true gesetzt.

timeoutFactor: Der timeoutFactor setzt die Zeit, wann der Token erneuert werden soll. 0.75 bedeutet, dass der Token nach 75% der eigentlichen Ablaufzeit erneuert wird.

sessionChecksEnabled: Gibt an, ob regelmäßig überprüft werden soll, ob der Benutzer noch gültig autorisiert ist.

clearHashAfterLogin: Gibt an, ob das Hash-Fragment nach der Anmeldung gelöscht werden soll.

redirectUri: Der Pfad zur Seite, auf die der Benutzer nach der erfolgreichen Authentifizierung weitergeleitet wird.

postLogoutRedirectUri: Der Pfad, zur der der Benutzer nach dem Abmelden weitergeleitet wird.

logoutUrl: Die URL für den Abmelde-Vorgang.

revocationEndpoint: Die URL, die zum Token-Widerruf-Endpunkt des Identity Servers führt, um beim Abmelden die Token zu widerrufen.

[42] [43] [44]

Nach dem die Konfigurationen vorgenommen und in den Service geladen wurden, startet der Richtige Login-Vorgang.

```
this.oauthService.configure(authCodeFlowConfig);
await this.oauthService.loadDiscoveryDocumentAndTryLogin();
if(! this.oauthService.hasValidAccessToken()){
    await this.oauthService.initLoginFlow();
}
```

Die Methode loadDiscoveryDocumentAndLogin lädt zuerst die Discovery-Dokumente vom Server herunter. Danach prüft die Methode, ob ein gültiger Token vorhanden ist und wenn ein gültiger Token vorhanden ist, wird der Benutzer automatisch als eingeloggt betrachtet.

Falls der Benutzer keinen gültigen Access-Token besitzt, wird die initLoginFlow-Mehtode aufgerufen, der den CodeFlow startet und den Benutzer zum Authorization-Server weiterleitet, um sich anzumelden.

Wenn der Benutzer dann einen gültigen Access-Token besitzt, werden die Benutzerdaten aus den Token ausgelesen. Für die Anwendung wird nur der Username und die UserId des Benutzers ausgelesen und gespeichert.

```
if(this.oauthService.hasValidAccessToken()){
    this.isAuthenticated.set(true);
    const claims = this.oauthService.getIdentityClaims();
    if (claims){
        const userName = claims[NAME_CLAIM];
        const userId = claims[ID_CLAIM];
        if(userName && userId){
            this.store.dispatch(new SetUser({guid: userId, name: userName}))
        }
    }
    this.router.navigate(['/home']);
}
```

Zum Schluss wird noch die automatische Token-Erneuerung aktiviert, damit sich der Access-Token immer automatisch erneuert bevor der Token abläuft.

```
this.oauthService.setupAutomaticSilentRefresh();
```

Falls während des Anmeldevorgangs ein Fehler auftritt und der Anmeldevorgang wird abgebrochen, wird der Benutzer zur CancelLoginPageComponent weitergeleitet. Dort hat er die Möglichkeit, sich neu Anzumelden.

```
}catch (error){
    this.router.navigate(['cancle'])
}
```

Die Logout Methode wird verwendet, um den Benutzer abzumelden und die Token ungültig zu machen.

```
public async logout(){
    await this.oauthService.revokeTokenAndLogout();
    this.store.dispatch(new SetUser(undefined));
    this.isAuthenticated.set(false);
  }
```

Die Methode revokeTokenAndLogout sendet zuerst eine Anfrage an den Identity Provider um die Token zu widerrufen, damit die Token nicht mehr für den Zugriff auf das System verwendet werden können. Danach wird der Benutzer auf die konfigurierte LogoutUrl weitergeleitet und komplett abgemeldet.

ChatService

Der Chat-Service wird verwendet, um die Chats des Benutzers zu verwalten und bietet die Methoden zum Hinzufügen, Löschen, Bearbeiten und Lesen der Chats an. Der Service kommuniziert durch eine REST-API mit dem Backend.

```
getUserChats(){
    const headers:HttpHeaders = this.store.selectSnapshot(AppState.httpHeader);
    return this.http.get<Chat[]>(url+'/api/Documents/chat', {headers})
  }
```

Die Methode getUserChat holt alle Chats des Benutzers durch einen HTTP-Request vom Backend.

```
addNewChat(){
    const headers:HttpHeaders = this.store.selectSnapshot(AppState.httpHeader);
    let chat: Chat = {
      id: 0,
      name: '',
      creationTime: undefined,
      userGUID: this.store.selectSnapshot(AppState.currentUser)?.guid!,
      queryItems: []
    }
    return this.http.post(url+'/api/Documents/chat',chat,{headers})
}
```

Die Methode addNewChat erstellt einen neuen Chat für den Benutzer. Dafür wird ein POST-Request an das Backend gesendet, bei dem im Request-Body ein Chat-Objekt mitgegeben wird, das nur Standardwerte enthält, außer der userGUID des eingeloggten Benutzers. Die Methode gibt das neu erstellte Chat-Objekt zurück, das der POST-Request zurückgibt.

```
deleteChat(chat: Chat){
    const headers:HttpHeaders = this.store.selectSnapshot(AppState.httpHeader);
    return this.http.delete(url+'/api/Documents/chat?id='+chat.id,{headers,observe :
    ↪  "response"});
}
```

Die deleteChat-Methode löscht einen bestimmten Chat eines Benutzers. Sie erhält das zu löschende Chat-Objekt als Parameter. Die Methode sendet einen DELETE-Request und übergibt dabei die ID des Chats.

```
updateChat(chat: Chat){
    const headers:HttpHeaders = this.store.selectSnapshot(AppState.httpHeader);
    return this.http.put<Chat>(url+'/api/Documents/chat',chat,
    ↪  {headers,observe:"body"});
}
```

Die updateChat-Methode bearbeitet einen bestimmten Chat des Benutzers. Sie erhält das überarbeitete Chat-Objekt als Parameter und sendet einen PUT-Request an die API. Das bearbeitete Chat-Objekt wird dabei im Request-Body mitgegeben. Die Methode gibt das vom Server zurückgegebene, aktualisierte Chat-Objekt zurück.

LLMService

Der LLMService wird nur verwendet um die verschiedenen LLM-Contents vom Backend zu holen, deshalb beinhaltet der Service auch nur eine Methode. Diese Methode heißt getLlms und holt alle LLM-Contents durch einen GET-Request vom Backend und gibt das Array von Llm-Objekten, die der Request zurückgibt, zurück.

```
getLlms(){
    const headers:HttpHeaders = this.store.selectSnapshot(AppState.httpHeader);
    return this.http.get<Llm[]>(url+"/api/Documents/llm", {headers})
}
```

MetaService

Der MetaService enthält nur die Methode getEmptyMetaObject, die nur ein neues Meta-Objekt mit Standardwerten zurückgibt.

```
getEmptyMetaObject(){
    const meta: Meta = {
      id: 0,
      title: '',
      fileName: '',
      fileType: '',
      format: '',
      creator: '',
      author: '',
      language: '',
      createDate: new Date(),
      modifyDate: new Date(),
      fileSize: 0,
      dir: ''
    }
    return meta;
  }
```

QueryService

QueryService wird verwendet, um ein Query zu erstellen oder zu updaten. Der Service erhält die Methode, die verwendet wird, um eine Frage zu beantworten und eine Methode, einen Query zu bearbeiten.

Die Methode sendMessage holt die Antwort einer Frage vom Backend und gibt das neu erstellte Query-Objekt und die Dauer der Abfrage zum Backend zurück.

Parameter:

msg: string = msg ist die Frage, auf die der Benutzer eine Anwort erhalten möchte.

llm: string = llm ist der vom Benutzer ausgewählte LLM-Content, der zum Beantworten der Frage verwendet wird .

searchDb: string = searchDb ist die ausgewählte Suchdatenbank, die der Benutzer zur Beantwortung der Frage verwenden möchte.

```
let searchType = "Lexical";
    if (searchDb === "milvus")
      searchType = "Vector";
```

Zuerst wird der richtige SearchType für die Query definiert. Wenn searchDb = „milvus“ ist, dann soll die Vektorsuche angewendet werden und searchType wird auf „Vector“ gesetzt. Andernfalls wird searchType auf „Lexical“ gesetzt, damit die lexikalische Suche angewendet wird.

```
this.countdownService.startTimer();
const response = await this.http.post(url+'/api/Documents/query', {
      id: 0,
      chatId: this.store.selectSnapshot(AppState.currentChat)?.id,
      question: msg,
      answer: "",
      llM_Content: llm,
      duration: 0,
      sendTime: new Date().toISOString(),
      searchType: searchType,
      meta: this.metaService.getEmptyMetaObject()
}, {headers}).toPromise();
```

```
13 const duration = this.countdownService.stopTimer();
14 return {data: response, duration};
```

Um die benötigte Zeit für die Beantwortung der Frage zu ermitteln, wird die Dauer des POST-Requests mithilfe des CountdownService gemessen. Der POST-Request enthält im Body ein neues Query-Objekt, das noch keine Antwort und Dauer enthält. Nachdem der POST-Request abgeschlossen ist, wird das aktualisierten Query-Objekt (inklusive der Antwort) sowie die gemessenen Dauer des Requests zurückgegeben.

```
1 updateQuery(updatedQuery: Query){
2     const headers:HttpHeaders = this.store.selectSnapshot(AppState.httpHeader);
3     return this.http.put<Query>(url+'/api/Documents/query',updatedQuery,
      ↪ {headers,observe:"body"});
4 }
```

Die updateQuery-Methode sendet einen PUT-Request an das Backend, um das Query-Objekt in der Datenbank zu ändern. Dafür wird im Request-Body das aktualisierte Query-Objekt mitgeschickt. Die Methode gibt anschließend das aktualisierte Query-Objekt zurück, das beim Request zurückgegeben wird.

CountdownService

Der CountdownService stellt die Funktionen einer Stoppuhr bereit. Der Service wird im Programm verwendet, um zu messen, wie lange das Backend benötigt, um eine Frage zu beantworten. Dazu bietet der Service zwei Methoden:

- startTimer() startet die Stoppuhr, indem das Attribut startTime auf den aktuelle Zeitpunkt gesetzt wird.
- stopTimer() erfasst den Endzeitpunkt und berechnet die vergangene Zeitspanne seit startTime in Millisekunden.

```
1 export class CountdownService {
2   startTime: any;
3
4   startTimer(){
5     this.startTime = new Date().getTime();
6   }
7
```

```
8     stopTimer(){
9       const endTime = new Date().getTime();
10      return endTime - this.startTime;
11    }
12  }
```

4.4.6 Komponenten

LoginPageComponent

Die LoginPage-Komponente hat nur eine Funktion. Beim Aufruf der Komponente wird die login-Methode des AuthService aufgerufen. Diese Komponente wird immer aufgerufen, wenn der Benutzer nicht authentifiziert ist und die authGuard-Funktion dem Benutzer den Zutritt zu einer bestimmten Route verweigert.

CancleLoginPage

Die CancleLoginPage-Komponente, ist ein Seite, auf die der Benutzer weitergeleitet wird, wenn beim Login-Vorgang etwas schief geht. Diese Seite informiert den Benutzer nur, dass er nicht eingeloggt ist und stellt einen „Login“-Button zur Verfügung, der die login-Methode des AuthService aufruft.

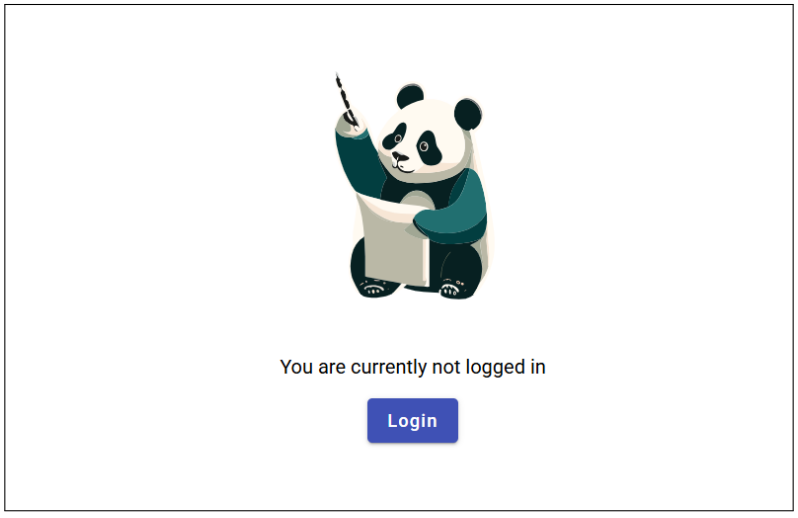


Abbildung 38: CancleLoginPageComponent Frontend

HomePageComponent

Die HomePage-Komponente ist die Hauptseite der Anwendung, auf die der Benutzer nur mit einem gültigen Access Token Zugriff hat. Diese Komponente ist verantwortlich die Komponenten ChatList, SearchPage und OutputPage auf eine Seite unterzubringen. Beim Aufruf der Komponente werden vom Backend alle LLM-Contents und die Chats vom eingeloggten Benutzer geholt.

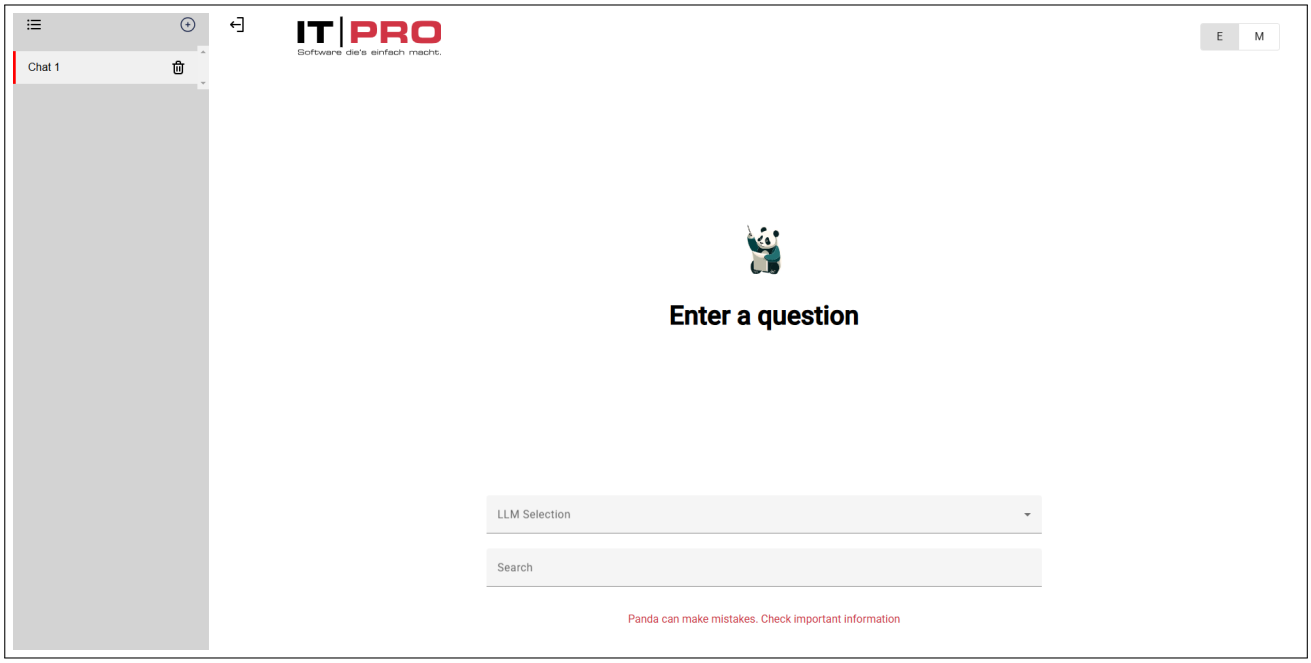


Abbildung 39: HomepageComponent Frontend

Code

```
public currentUser$ = this.store.select(AppState.currentUser);
showChatList = true;
```

currentUser$ ist ein Observable, der den eingeloggten User speichert.

showChatList speichert, ob die Chatliste angezeigt werden soll.

```
ngOnInit() {
    if (!this.currentUser$){
      this.router.navigate(['/login']);
    }
    this.store.dispatch(new GetLlmOptions());
    this.getUserChats();
}
```

Beim Aufruf der Komponente, wird zuerst überprüft, ob der eingeloggte User gespeichert wird. Wenn dies nicht der Fall sei, wird der Benutzer wieder zur Login-Seite weitergeleitet. Wenn jedoch der eingeloggt Benutzer richtig gespeichert wurde, werden die LLM-Options vom Store geholt und die getUserChats-Methode aufgerufen.

```
toggleChatList(){
    this.showChatList = !this.showChatList;
}
```

Wenn diese Methode aufgerufen wird, ändert sich der Wert von showChatList. Sie wird immer ausgeführt, wenn der Button zum Anzeigen oder Ausblenden der Chatliste in der ChatListComponent oder der OutputPageComponent angeklickt wird.

```
getUserChats(){
    this.store.dispatch(new FetchUserChats()).subscribe({
      next: () => {
        this.store.dispatch(new SetFirstChatAsCurrent());
      },
      error: (error) => {
        console.log("Error while fetching chats: " + error);
      }
    });
}
```

Diese Methode holt durch den Aufruf der FetchUserChats-Action alle Chats des eingeloggten Benutzers vom Backend. Wenn die Ausführung der Action erfolgreich war, wird noch die SetFirstChatAsCurrent-Action im State aufgerufen, damit der neueste Chat direkt ausgewählt wird.

SearchPageComponent

Die SearchPage-Komponente ist ein Teil der Hauptseite und enthält Eingabefelder für die Suche. Die Komponente beinhaltet eine Dropdown-Liste, in der der Benutzer einen der vom Backend bereitgestellten LLM-Contents auswählen kann. Außerdem gibt es ein Texteingabefeld, in das der Benutzer seine Frage eingibt. Die Eingabefelder sind in einem Formular eingebunden, das bei einem Submit die sendMessage-Methode aufruft, um die Frage zu beantworten.

Abbildung 40: Input-Felder Frontend

Code

```
@Select(AppState.llmOptions) llmOptions$!: Observable<Llm[]>;
@Select(AppState.currentChat) currentChat$!: Observable<Chat>;
slectedOption: Llm = {} as Llm;
searchtext: string = '';
```

llmOptions$ ist ein Observable, der eine Liste von allen LLM-Optionen speichert.
currentChat$ ist ein Observable, der die ganze Zeit den derzeit ausgewählten Chat speichert.
slectedOption speichert, das Llm-Objekt, das in der Dropdown-Liste gerade ausgewählt ist.
searchtext ist die Frage, die im Texteingabefeld steht.

Die wichtigste Methode in der Komponente ist die sendMessage-Methode, die beim Submit des Eingabeformulars aufgerufen wird. In der Methode werden die Funktionen aufgerufen, um die Frage zu Beantworten und den erstellten Query im Backend und im State richtig zu speichern.

```
if (!this.selectedOption.content) {
      this.llmOptions$.subscribe(llmOptions => {
        if (llmOptions.length > 0){
          const defaultOption = llmOptions.filter(llm => llm.content == '')[0];
```

```
          if (defaultOption){
            this.selectedOption = defaultOption;
          }else{
            this.selectedOption = llmOptions[0];
          }
        }
      })
}
```

Zu Beginn wird überprüft, ob der Benutzer bereits eine LLM-Option ausgewählt hat. Falls nicht, wird aus der llmOptions-Liste der Standard-LLM ohne Content gewählt, sofern der Standard-LLM in der Liste vorhanden ist. Ist der Standard-LLM nicht vorhanden, wird selectedOption auf das erste Element der Liste gesetzt. Falls die Liste leer ist, bleibt selectedOption ein leeres Objekt, wie es zu Beginn initialisiert wurde.

```
if (this.searchtext.length > 0 && this.selectedOption) {
    const newQuery: Query = {
        id: 0,
        chatId: 0,
        question: this.searchtext,
        answer: "",
        llmContent: this.selectedOption.content,
        sendTime: new Date(),
        searchType: '',
        duration: 0,
        meta: this.metaService.getEmptyMetaObject()
    };
    this.store.dispatch(new AddQuery(newQuery));
```

Nur wenn eine Frage im Texteingabefeld eingegeben wurde, läuft die Methode weiter. Zuerst wird ein neues Query-Objekt mit den bereits vorhandenen Informationen erstellt und durch die AddQuery-Action im State hinzugefügt.

```
const response: {data: any, duration: number} = await
↪  this.queryService.sendMessage(msg, this.selectedOption.content, searchDB!);

if (response) {
    if (response.data){
        if (response.data.answer.length > 0) {
```

```
            newQuery.answer = response.data.answer;
        } else {
            newQuery.answer = "No answer could be found!";
        }
        newQuery.chatId = response.data.chatId;
        newQuery.id = response.data.id;
        newQuery.duration = response.duration;
        newQuery.meta = this.getMetaData(response.data.meta);
        newQuery.searchType = response.data.searchType;
        this.store.dispatch(new UpdateQuery(newQuery));
    }
} else {
    newQuery.answer = "No answer could be found!";
    this.store.dispatch(new UpdateQuery(newQuery));
}
```

Als Nächstes wird die sendMessage-Methode des QueryService aufgerufen, um die Frage zu beantworten, den Query in der Datenbank zu speichern und die Dauer des Requests zu erhalten. Wenn die sendMessage-Methode eine gültige Antwort zurückgibt, werden die Attribute des neuen Query-Objekts, mit den Werten aus dem Response aktualisiert. Zum Schluss wird noch die UpdateQuery-Action aufgerufen, um den Query im State zu aktualisieren.

OutputPageComponent

Die OutputPage-Komponente besteht aus zwei Teilen. Im oberen Teil der Komponente sind verschiedene Button und das Logo der ITPRO und im unteren Teil wird der Chat angezeigt.

Im oberen Teil befindet sich ganz links der Button, um die Chatliste anzuzeigen und direkt daneben der Button, um einen neuen Chat zu erstellen. Diese Buttons werden jedoch nur angezeigt, wenn die Chatliste ausgeblendet ist. Wenn jedoch die Chatliste geöffnet ist, befindet sich der „Abmelde“-Button an dieser Stelle. Ansonsten befindet sich der Abmelde-Button direkt rechts neben dem Button zum erstellen eines neuen Chats. Neben den Abmelde-Button ist das Logo der ITPRO. Ganz Rechts im oberen Teil ist der Toggle, mit dem die Art der Beantwortung der Frage ausgewählt werden kann.

Abbildung 41: Oberer Teil OutputPage Komponente Frontend

Im unteren Bereich der Komponente wird der ausgewählte Chat angezeigt. Zuerst wird die Frage und darunter die Antwort angezeigt. Beide werden in getrennten Kästchen angezeigt. Die Frage befindet sich in einem hellgrauen Kästchen und ist rechtsbündig ausgerichtet, während die Antwort zusammen mit dem Namen und Autor des Dokuments, in dem die Antwort gefunden wurde, der Art der Suche und der Antwortzeit in einem dunkelgrauen Kästchen angezeigt wird, das linksbündig ausgerichtet ist.

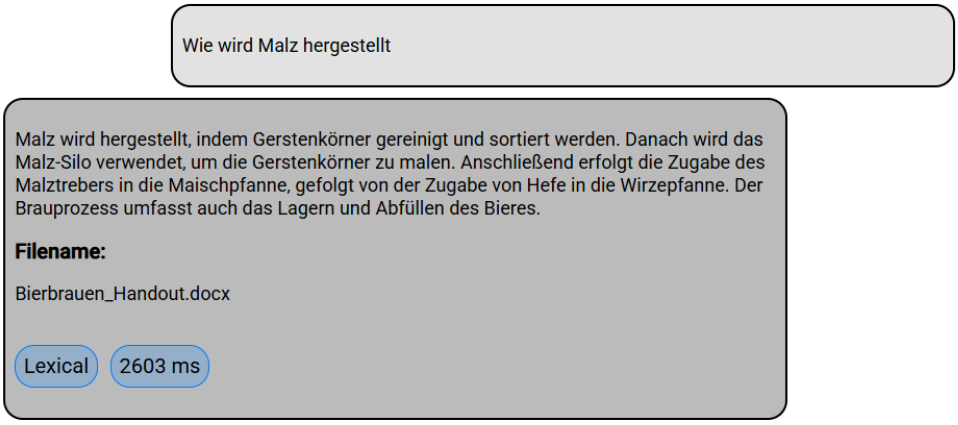


Abbildung 42: Ausgabe Frontend

Wenn jedoch der Chat keine Fragen beinhaltet und daher leer ist, wird das PANDA-Logo gemeinsam mit den Text „Enter a question“ angezeigt.

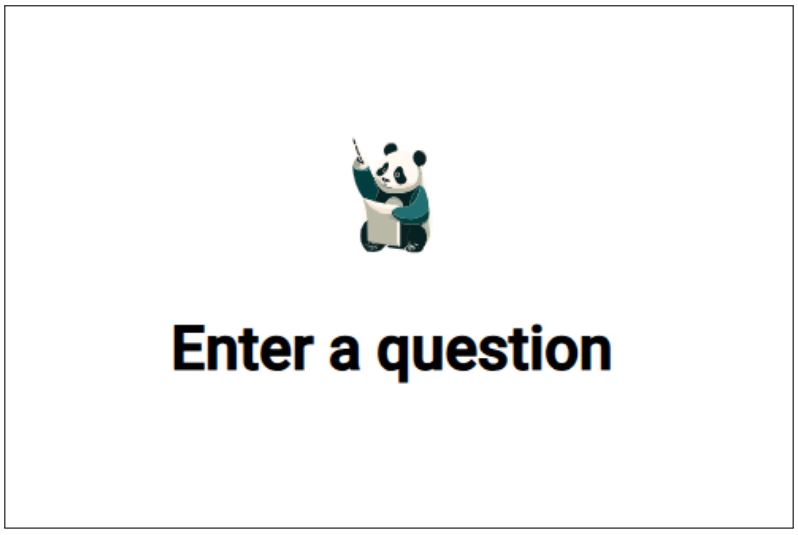


Abbildung 43: Leerer Chat Ansicht

Code

Zu Beginn werden die Attribute definiert.

```
@Select(AppState.chatQueries) queryList$!: Observable<Query[]>;
@Input() showChatList:boolean = false;
@Output() toggleChat = new EventEmitter<void>();
@ViewChild('outputcontainer') private scrollContainer!: ElementRef;
```

queryList$ ist eine Observable-Liste, die vom AppState, die Queries vom angezeigten Chat abruft. Wenn sich chatQueries im AppState ändert, wird die Liste automatisch mitaktualisiert.
showChatList wird benötigt, um abzufragen, ob der Button zum Anzeigen des Chats sichtbar sein soll. Der Wert wird von der HomePageComponent übergegeben.
toggleChat ist ein EventEmitter, mit dem die Methode toggleChatList in der HomePageComponent aufgerufen werden kann.
scrollContainer ist ein DOM-Element auf das über ViewChild auf das Element mit der Referenz „outputcontainer“ zugegriffen wird.

```
public set db(db:string | undefined){
    if(db){
      this.store.dispatch(new SetDb(db))
    }else{
      console.error("No Db found")
    }
  }
```

Das ist ein Setter für die Variable db. Wenn der Setter aufgerufen wird, wird zuerst überprüft, ob db nicht undefined ist. Wenn db nicht undefined ist, wird der neue Wert im Store gespeichert. Wenn nicht, wird eine Fehlermeldung ausgegeben

```
private scrollToBottom(): void {
    try {
      this.scrollContainer.nativeElement.scrollTop =
      ↪  this.scrollContainer.nativeElement.scrollHeight;
    } catch(err) {
      console.log("Error while scrolling to bottom")
```

```
    }
}
```

Diese Methode, ist dafür verantwortlich, dass im scrollContainer automatisch ganz nach unten gescrollt werden soll, wenn die Methode aufgerufen wird.

```
ngAfterViewChecked() {
    this.scrollToBottom();
}
```

Die Methode ist ein Angular-Lebenszyklus-Hook und wird jedes Mal aufgerufen, wenn sich die Ansicht der Komponente ändert. Innerhalb der Methode wird die scrollToBottom-Methode aufgerufen, die dafür sorgt, dass bei jeder Änderung der Chat automatisch nach unten gescrollt wird. So wird sichergestellt, dass immer, wenn eine neue Frage mit Antwort ausgegeben wird, automatisch zur letzten Frage gescrollt wird.

```
toggleChatList(){
    this.toggleChat.emit();
}
```

Die toggleChatList-Methode wird immer aufgerufen, wenn der Button zum Anzeigen der Chatliste geklickt wird. Dabei wird das toggleChat-Event emittiert und dadurch wird die toggleChatList-Methode in der HomePageComponent aufgerufen.

```
addChat(){
    this.store.dispatch(new AddChat())
}
```

Diese Methode wird aufgerufen, wenn der Button zum erstellen eines neuen Chats angeklickt wird. Es wird die AddChat-Methode vom State aufgerufen, damit ein neuer Chat erstellt wird.

```
changeDb(event:any){
    this.db = event.value;
}
```

Diese Methode wird aufgerufen, wenn der Toggle zum Auswählen der Suchdatenbank ausgelöst wird und ruft den Setter für db auf.

```
logout() {
    this.authService.logout();
  }
```

Ist die Methode, um sich Abzumelden und wird aufgerufen, wenn auf den Abmelde-Button geklickt wird. Die Methode ruft die logout-Methode vom AuthService auf.

ChatListComponent

Die Chatlist-Komponente ist eine ausklappbare Sidebar zur Verwaltung der Chats des Benutzers. In der oberen linken Ecke befindet sich ein Button zum Ausblenden der Sidebar, und in der oberen rechten Ecke gibt es einen Button zum Erstellen eines neuen Chats.

Darunter werden die Chats des Benutzers aufgelistet. Bei jedem Chat ist der Name ersichtlich, der durch einen Doppelklick direkt bearbeitet werden kann. Rechts daneben befindet sich ein Button zum Löschen des jeweiligen Chats.

Abbildung 44: Chatliste Frontend

Code

```
@Output() toggleChat = new EventEmitter<void>;
@Select(AppState.userChats) chats$!: Observable<Chat[]>;
@Select(AppState.currentChat) currentChat$!: Observable<Chat>;
isEditing = false;
```

toggleChat ist ein EventEmitter, mit dem die Methode toggleChatList() in der übergeordneten HomePageComponent aufgerufen werden kann. Dies wird verwendet, um die Chatliste auszublenden.

chats$ ist ein Observable, das die Liste der aktuellen Chats des Benutzers aus dem AppState abruft. Diese Liste wird automatisch aktualisiert, wenn sich der Zustand der Anwendung verändert.

currentChat$ ist ein Observable, der den aktuell ausgewählten Chat speichert.

isEditing ist ein Boolean, das angibt, ob das Input-Element, welches den Chatnamen anzeigt,

bearbeitet werden kann oder auf readonly gesetzt ist.

```
toggleChatList(){
    this.toggleChat.emit();
}
```

Diese Methode wird bei Klick auf den Button zum Schließen der Chatliste aufgerufen und lösten den toggleChat EventEmitter aus.

```
addChat(){
    this.store.dispatch(new AddChat())
}
```

Diese Methode wird aufgerufen, wenn der Button zum Hinzufügen eines neuen Chats geklickt wird. Sie fügt einen neuen Chat zum State hinzu, indem die AddChat-Aktion ausgelöst wird.

```
listItemSelected(chat: Chat){
    if (chat){
      this.store.dispatch(new SetChat(chat));
    }
}
```

Diese Methode wird aufgerufen, wenn der Benutzer auf einen Chat in der Liste klickt. Der ausgewählte Chat wird im State gespeichert, indem die SetChat-Aktion ausgelöst wird.

```
deleteChat(chat: Chat){
    this.store.dispatch(new DeleteChat(chat));
}
```

Diese Methode wird aufgerufen, wenn der Benutzer auf den Chat-Löschen-Button klickt. Sie löst die DeleteChat-Aktion aus, die den Chat aus dem State löscht.

```
enableEditing() {
    this.isEditing = true;
  }
```

Diese Methode wird aufgerufen, wenn das Input-Element doppelt angeklickt wird, um den Namen des Chats zu bearbeiten. Sie setzt isEditing auf true, sodass das Eingabefeld editierbar wird.

```
editName(chat: Chat){
    this.store.dispatch(new UpdateChat(chat));
    this.isEditing = false;
}
```

Diese Methode wird aufgerufen, wenn das Input-Element den Fokus verliert, weil der Benutzer es verlässt. Dabei wird die UpdateChat-Aktion ausgelöst, um den Chat im State zu aktualisieren. Außerdem wird isEditing auf false gesetzt, sodass das Input-Element auf readonly gesetzt wird.

4.4.7 Route-Guard

Route-Guards in Angular schützen Routen, indem sie vor dem Laden einer Route prüfen, ob der Benutzer die erforderlichen Berechtigungen hat. Route-Guards können den Zugriff auf bestimmte Seiten für einen Benutzer erlauben oder verweigern. Es gibt verschiedene Route-Guards in Angular, die Anwendung nutzt jedoch nur den CanActivateFn Guard. CanActivateFn prüft, ob eine Route vom Benutzer betreten werden darf oder nicht.

```
export const authGuard: CanActivateFn = (route, state) => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (authService.hasValidAccessToken()) {
    return true;
  } else {
    router.navigate(['login']);
    return false;
  }
};
```

In der Funktion wird mithilfe dem AuthService geprüft, ob der Benutzer einen gültigen Access Token besitzt. Wenn der Access Token gültig ist, wird der Zugriff gestatten, wenn jedoch der Access Token ungültig ist, wird der Zugriff verweigert und der Benutzer wird zur 'login'-Route weitergeleitet.

Es muss auch festgelegt werden, bei welchen Routen die authGuard-Funktion aufgerufen wird. Dies geschieht in der app-routing.module.ts-Datei, in der alle Routen definiert sind.

```
const routes: Routes = [
  { path: 'login', component: LoginPageComponent},
  { path: 'cancle', component: CancleLoginPageComponent},
  { path: 'home', component: HomePageComponent, canActivate: [authGuard]},
  { path: '**', redirectTo: 'login'},
];
```

Wie im Code ersichtlich, wird die authGuard-Funktion nur für den Zugriff auf die HomePage-Component aufgerufen und alle anderen Routen sind ohne Einschränkungen zugänglich.

[45]

4.4.8 Silent Refresh

Die Anwendung verwendet Silent Refresh, damit sich der Benutzer nicht ständig anmelden muss und der Access Token automatisch im Hintergrund erneuert wird, bevor er abläuft. Im Code mussten einige Maßnahmen vorgenommen werden, damit Silent Refresh richtig funktioniert. In der login-Methode im AuthService gibt es in der AuthConfig einige wichtige Konfigurationen, die für das Silent Refresh benötigt werden.

```
const authCodeFlowConfig: AuthConfig = {
      scope: 'openid profile IdentityServerApi PandaApi offline_access',
      silentRefreshRedirectUri: `${window.location.origin}/silent-refresh.html`,
      useSilentRefresh: true,
      timeoutFactor: 0.75,
      sessionChecksEnabled: true,
};
```

scope: Der Scope „offline_access“ ist sehr wichtig für den Silent Refresh, da ansonsten kein Refresh Token zusammen mit dem Access Token übergeben wird und dadurch ein erfolgreicher Silent Refresh nicht möglich wäre.
useSilentRefresh: true Aktiviert Silent Refresh, sodass der Token automatisch erneuert wird.
sessionChecksEnabled: true Dadurch wird die Session des Benutzers ständig überwacht, um zu erkennen, ob sie noch gültig ist.
timeoutFactor: 0.75 Das ist eine zusätzliche Konfiguration. Normalerweise wird Silent Refresh durchgeführt, wenn der Token abgelaufen ist. Mit timeoutFactor kann festgelegt werden, wann

Silent Refresh ausgelöst werden soll. Da der Wert auf 0.75 gesetzt ist, wird Silent Refresh nach 75% der Lebenszeit des Tokens ausgeführt.

silentRefreshRedirectUri: Das ist der Pfad, an den der Benutzer nach der Silent Refresh Anforderung weitergeleitet wird. Hier bedeutet es, dass nachdem der Silent Refresh erfolgreich durchgeführt wird und der Access Token erneuert wurde, der Benutzer zur silent-refresh.html weitergeleitet wird und der Server gibt dabei den neuen Access Token im URL-Fragment mit.

Da der Access Token im Hintergrund automatisch erneuert wird, ohne dass der Benutzer eingreifen muss, wird die silent-refresh.html in einem unsichtbaren iFrame geladen.

Die silent-refresh.html sieht wie folgt aus:

```
<!DOCTYPE html>
<html>
<head>
  <title>Silent Refresh</title>
  <script>
    function handleSilentRefresh() {
    var hash = window.location.hash.substr(1);
      var result = hash.split('&').reduce(function (res, item) {
        var parts = item.split('=');
        res[parts[0]] = parts[1];
        return res;
      }, {});
      window.parent.postMessage(result, window.location.origin);
    }
    window.onload = handleSilentRefresh;
  </script>
</head>
<body>
<p>Loading...</p>
</body>
</html>
```

Diese Seite wird nur verwendet, um den erneuerten Access Token aus der URL zu extrahieren und an die Hauptanwendung zurückzugeben.

Wenn silent-refresh.html vollständig geladen ist, wird die handleSilentRefresh-Methode aufgerufen. Dort wird zuerst der Fragmentteil der URL ausgelesen. Danach werden die verschiedenen

Token-Informationen ausgelesen und in Key-Value Paare umgewandelt. Zum Schluss werden die Token-Informationen an die Hauptanwendung zurückgegeben.

Weiter gibt es in der login-Methode im AuthService noch den wichtigsten Befehl.

```
1 this.oauthService.setupAutomaticSilentRefresh();
```

Durch diesen Befehl wird die automatische Durchführung des Silent Refresh eingerichtet, sodass der Access Token im Hintergrund automatisch erneuert wird.

4.5 Backend

4.5.1 Aufteilung

Das Backend des RAG-Systems besteht aus zwei individuell ausführbaren Programmen.

Der DocumentDissector ist für die automatische Überwachung eines Ordners zuständig. Alle Dokumente in diesem Ordner folgen derselben Prozedur: Die Informationen werden extrahiert, in die richtige Form für die Suchdatenbanken transformiert und dann indiziert. Dies wird auch bei der Veränderung der Dateien neu durchgeführt.

Die Businesslogik wird gemeinsam mit der REST-API in der Solution „panda-backend“ ausgeführt. Sie enthält zwei Services, von denen einer für die Verwaltung der Chats im Frontend zuständig ist und EF Core verwendet. Der zweite Service ist für die Beantwortung einer Benutzerfrage zuständig. Dafür wird eine Kommunikation mit der OpenAI-API hergestellt und Wörter und Synonyme werden zu Vektor-Embeddings oder Elasticsearch-Querys zusammengebaut.

4.5.2 Datenbeschaffung

DoccumentDissector

Um die mit FSCrawler entstandenen Probleme bezüglich Dokumentengröße und Datenbankunabhängigkeitzu lösen, wurde das Datenbeschaffungsprogramm *DocumentDissector* geschrieben.

Dieses Datenbeschaffungsprogramm lässt sich in folgende sechs Schritte unterteilen:

1. Verzeichnisscan
2. Prüfung auf Veränderungen
3. Extrahieren des in der Datei enthaltenen Textes
4. Extrahieren der Metadaten der Datei
5. Text in Chunks aufteilen
6. Indizieren in die Datenbanken

Das Hauptprogramm

Im Hauptprogramm werden hauptsächlich die Instanzen aller anderen Services verwaltet und aufgerufen.

```
// Load configuration settings from appsettings.json
IConfiguration configuration = new ConfigurationBuilder()
    .SetBasePath(Directory.GetCurrentDirectory())
    .AddJsonFile("appsettings.json", false, false)
    .Build();

// Create necessary instances of dependency classes
FileCrawler crawler = new FileCrawler(); // Responsible for scanning files in the directory
ContentExtractor extractor = new ContentExtractor(configuration); // Extracts content from files
Chunker chunker = new Chunker(configuration); // Splits extracted content into smaller chunks
IDBClient elasticClient = new ElasticClient(configuration); // Interface for Elasticsearch database client
IDBClient milvusClient = new MilvusClient(configuration); // Interface for Milvus vector database client

// Instantiate services for indexing and deletion
IndexerService indexer = new IndexerService(extractor, chunker, elasticClient, milvusClient);
DeletionService deletionService = new DeletionService(elasticClient, milvusClient);
```

Außerdem wird ein initialer Scan mit dem IndexVerifier ausgeführt.

```
// On program startup: Verify synchronization between file system and index
IndexVerifier verifier = new IndexVerifier(crawler, elasticClient, milvusClient, indexer);
await verifier.VerifyAsync(configuration);
```

Zuletzt wird noch der FileMonitor initialisiert.

```
// Get the directory path to monitor from configuration
string directoryToWatch =
    configuration.GetValue<string>("AppConfig:DirectoryBasePath");
```

```
if (string.IsNullOrWhiteSpace(directoryToWatch) ||
    !Directory.Exists(directoryToWatch))
{
    Console.WriteLine("Monitoring directory not found or invalid.");
    return;
}

// Create and start the FileSystemWatcher to monitor directory changes
FileMonitor monitor = new FileMonitor(directoryToWatch, indexer)
{
    // Callback function for handling file deletions
    OnDeleteCallback = fileName => deletionService.Delete(fileName)
};
monitor.Start();
```

IndexVerifier

Der IndexVerifier ist dafür zuständig, dass nach einem Neustart des Programms das Verzeichnis initial auf Veränderungen zur Datenbank überprüft wird. Als erster Schritt werden alle Dateien im Verzeichnis mithilfe des FileCrawlers geholt, um das ModifyDate aller Dateien mit dem in der Datenbank gespeicherten kontrollieren zu können.

```
// Retrieve all files from the monitored directory.
var files = _crawler.GetFilePaths(configuration);

foreach (string file in files)
{
    DateTime currentModifyDate = File.GetLastWriteTime(file);
    string fileName = Path.GetFileName(file);

    // Search for existing index entries in Elasticsearch based on file name.
    var searchResults = await
        _elasticClient.SearchDocumentsByFileNameAsync(fileName);
    bool needsIndexing = true;

    if (searchResults != null && searchResults.Any())
    {
        foreach (var doc in searchResults)
```

```
        {
            // Check if the modification date matches the indexed version.
            if (doc.Meta.ModifyDate.HasValue && doc.Meta.ModifyDate.Value ==
            ↪ currentModifyDate)
            {
                needsIndexing = false;
                break;
            }
        }
    }

    // If the file is missing or outdated in the index, reprocess it.
    if (needsIndexing)
    {
        Console.WriteLine($"[Verifier] Indexing missing or outdated for: {file}");
        await _indexer.ProcessFileAsync(file);
    }
}
```

Hat sich etwas verändert, wird für die jeweilige Datei die Methode *ProcessFileAsync* des IndexerService aufgerufen.

Zuletzt werden noch einmal alle in der Datenbank gespeicherten Einträge überprüft, ob sie überhaupt noch im Dateisystem existieren, ansonsten werden sie aus der Datenbank gelöscht.

```
// Retrieve all indexed documents from Elasticsearch.
var allDocs = await _elasticClient.GetAllIndexedDocumentsAsync();
if (allDocs != null)
{
    foreach (var doc in allDocs)
    {
        // Check if the indexed file still exists in the file system.
        bool fileExists = files.Any(f =>
            Path.GetFileName(f).Equals(doc.Meta.FileName,
            ↪ StringComparison.OrdinalIgnoreCase));

        // If the file no longer exists, remove its index entry.
        if (!fileExists)
        {
```

```
            Console.WriteLine($"[Verifier] Deleting orphaned index entry for:
            ↪ {doc.Meta.FileName}");
            try
            {
                _elasticClient.DeleteEntries(doc.Meta.FileName);
                _milvusClient.DeleteEntries(doc.Meta.FileName);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

FileMonitor

In der Klasse *FileMonitor* werden die Konfigurationen des *FileSystemWatchers* vorgenommen, welcher das angegebene Verzeichnis andauernd auf Änderungen überprüft und je nach Änderung die passende Methode dazu aufruft.

```
public FileMonitor(string directory, IndexerService indexer)
{
    _indexer = indexer;
    _watcher = new FileSystemWatcher(directory)
    {
        IncludeSubdirectories = true,
        Filter = "*.*",
        NotifyFilter = NotifyFilters.FileName | NotifyFilters.LastWrite
    };

    _watcher.Created += async (s, e) =>
    {
        Console.WriteLine($"[Created] {e.FullPath}");
        await _indexer.ProcessFileAsync(e.FullPath);
    };

    _watcher.Changed += async (s, e) =>
```

```
18      {
19          Console.WriteLine($"[Changed] {e.FullPath}");
20          await _indexer.ProcessFileAsync(e.FullPath);
21      };
22
23      _watcher.Deleted += (s, e) =>
24      {
25          Console.WriteLine($"[Deleted] {e.FullPath}");
26          string fileName = Path.GetFileName(e.FullPath);
27          OnDeleteCallback?.Invoke(fileName);
28      };
29
30      _watcher.Renamed += async (s, e) =>
31      {
32          Console.WriteLine($"[Renamed] {e.OldFullPath} -> {e.FullPath}");
33          string fileNameOld = Path.GetFileName(e.OldFullPath);
34          OnDeleteCallback?.Invoke(fileNameOld);
35          await _indexer.ProcessFileAsync(e.FullPath);
36      };
37  }
```

Wird eine Datei neu erstellt oder ändert sich an ihrem Inhalt etwas, wird in beiden Fällen die *ProcessFileAsync* Methode des *IndexerService* aufgerufen. Wird eine Datei aber gelöscht oder umbenannt, wird sie komplett aus der Datenbank entfernt, da sie nicht mehr anhand des Dateinamens gefunden werden kann. Nach dem Umbenennen wird sie aber wieder komplett neu indiziert.

Um den *FileSystemWatcher* zu starten bzw. zu beenden, gibt es die Methoden *Start* und *Stop*:

```
1  public void Start() => _watcher.EnableRaisingEvents = true;
2  public void Stop() => _watcher.EnableRaisingEvents = false;
```

IndexerService

Die *ProcessFileAsync* Methode bildet die Hauptfunktion der *IndexerService* Klasse. Bevor, wie der Name schon erahnen lässt, Dokumente in die Datenbanken indiziert werden, werden die Dateien aber zuerst durch die Funktionalitäten des *MetaExtractors*, des *ContentExtractors* sowie des *Chunkers* vorverarbeitet. Danach werden alle Chunks in die Datenbanken geschrieben:

```
    public async Task ProcessFileAsync(string file)
{
    try
    {
        // Short delay to ensure the file has been fully written before processing.
        await Task.Delay(100);

        // Retrieve the last modification date of the file from the file system.
        DateTime currentModifyDate = File.GetLastWriteTime(file);

        // Extract metadata (e.g., file name, type, etc.) and update the
        ↪ modification date.
        Meta metaData = MetaExtractor.GetMeta(file);
        metaData.ModifyDate = currentModifyDate;

        // Extract the text content from the file.
        string content = await _extractor.GetText(file);

        // Split the extracted content into smaller chunks.
        string[] chunks = _chunker.ChunkRecursive(content);
        int idCounter = 0;

        foreach (string chunk in chunks)
        {
            metaData.Id = idCounter++;
            Document document = new Document { Content = chunk, Meta = metaData };

            // Add the chunk to Elasticsearch.
            _elasticClient.AddEntry(document);

            // Add the chunk to Milvus.
            _milvusClient.AddEntry(document);
        }

        Console.WriteLine($"[Indexed] {metaData.FileName}");
    }
    catch (Exception ex)
```

```
37      {
38          Console.WriteLine($"[ERROR] Error processing file '{file}': {ex.Message}");
39      }
40  }
```

DeletionService

Die Klasse *DeletionService* ist konträr zum *IndexerService* dafür da, Einträge wieder aus allen Datenbanken zu entfernen.

```
1   public void Delete(string fileName)
2   {
3       try
4       {
5           // Remove indexed entries from Elasticsearch.
6           _elasticClient.DeleteEntries(fileName);
7
8           // Remove indexed entries from Milvus.
9           _milvusClient.DeleteEntries(fileName);
10      }
11      catch (Exception ex)
12      {
13          Console.WriteLine($"[ERROR] Error deleting file '{fileName}':
            ↪ {ex.Message}");
14      }
15  }
```

FileCrawler

In der für den Verzeichnisscan zuständigen Klasse *FileCrawler* ist nur die Funktionalität implementiert, welche eine Liste von Strings retourniert. Diese Strings sind die Dateipfade aller, der im übergebenen Verzeichnis enthaltenen Dateien. Durch das Überladen der Methode *GetFilePaths* ist es möglich, entweder einen beliebigen Verzeichnispfad anzugeben, welcher gescannt werden soll, oder den in den Appsettings definierten Pfad zu verwenden, um darin zu scannen.

```
1   public class FileCrawler
2   {
3       public List<string> GetFilePaths(IConfiguration configuration)
```

```
4       {
5           return
            ↪ GetFilePaths(configuration.GetValue<string>("AppConfig:DirectoryBasePath"));
6       }
7
8       public List<string> GetFilePaths(string directory)
9       {
10          if (directory == null) throw new InvalidOperationException("No base filepath
            ↪ set");
11
12          if (!Directory.Exists(directory)) throw new
            ↪ InvalidOperationException($"Directory {directory} does not exist");
13
14          return [.. Directory.GetFiles(directory, "*", SearchOption.AllDirectories)];
15      }
16  }
```

Abgesehen vom Überladen der Methode und dem Exception-Handling ist die gesamte Funktionalität in Zeile 14 beim Return angeordnet. Die Methode *GetFiles*, die dem Paket *System.IO.Directory* entspringt, ruft alle Dateien und Unterordner des im ersten Parameter übergebenen Pfades auf. Im zweiten Parameter kann ein Suchmuster übergeben werden, um die Suche zu spezifizieren. Da alle Dateien und Unterordner ermittelt werden sollen, wird hier ein Stern übergeben. Der letzte Parameter sorgt dafür, dass auch in allen Unterordnern gesucht wird. Neben der Option „AllDirectories“ gibt es auch noch „TopDirectoryOnly“, was die Suche auf das angegebene Verzeichnis ohne die Unterverzeichnisse beschränkt. Die zwei Punkte und die eckigen Klammern sind eine Kurzschreibweise, welche ein angegebenes Array in die richtige Liste verwandelt. Diese Syntax nennt man auch Spread-Syntax.

MetaExtractor

Die Klasse *MetaExtractor* wird benötigt, um die Metadaten der Dateien zu extrahieren. Dafür wird das Tool *Exiftool* verwendet, welches Metadaten aus fast allen Dateitypen in einem einheitlichen Format liefert. Siehe mehr im Kapitel Exiftool.

In der einzigen von außen aufrufbaren Methode *GetMeta* wird nur der Dateipfad übergeben, anhand dessen dann alle relevanten Metadaten extrahiert werden und als Meta-Objekt zurückgegeben werden.

```
public static Meta GetMeta(string filePath)
{
    // Get the metadata as a string from the file using exiftool.
    var metadataString = GetMetadataString(filePath);

    // Create a new Meta object and populate its properties by extracting specific fields from the metadata string.
    var meta = new Meta
    {
        Title = GetMetadataWithRegex(metadataString, "Title"),
        FileName = GetMetadataWithRegex(metadataString, "File Name"),
        FileType = GetMetadataWithRegex(metadataString, "File Type"),
        Format = GetMetadataWithRegex(metadataString, "Format"),
        Creator = GetMetadataWithRegex(metadataString, "Creator"),
        Author = GetMetadataWithRegex(metadataString, "Author"),
        Language = GetMetadataWithRegex(metadataString, "Language"),
        // Parse the creation and modification dates from the corresponding metadata fields.
        CreateDate = ParseDateFromString(GetMetadataWithRegex(metadataString, "File Creation Date/Time")),
        ModifyDate = ParseDateFromString(GetMetadataWithRegex(metadataString, "File Modification Date/Time")),
        // Parse the file size from the metadata.
        FileSize = ParseFileSizeFromString(GetMetadataWithRegex(metadataString, "File Size")),
        Dir = GetMetadataWithRegex(metadataString, "Directory")
    };

    // If CreateDate is null, try an alternative field "Create Date".
    meta.CreateDate ??= ParseDateFromString(GetMetadataWithRegex(metadataString, "Create Date"));
    // If ModifyDate is null, try an alternative field "Modify Date".
    meta.ModifyDate ??= ParseDateFromString(GetMetadataWithRegex(metadataString, "Modify Date"));

    return meta;
}
```

Wie im Code zu sehen, wird zuerst die private Methode *GetMetadataString* aufgerufen.

```
1  private static string GetMetadataString(string filePath)
2  {
3      // Create a new process to run exiftool.
4      Process process = new Process();
5      process.StartInfo.RedirectStandardOutput = true;
6      process.StartInfo.FileName = exifToolPath;
7      // The -n argument is used to output all numerical values in unformatted form
       ↪  without units
8      process.StartInfo.Arguments = $"-n \"{filePath}\"";
9      process.StartInfo.UseShellExecute = false;
10
11     // Start the process.
12     process.Start();
13
14     // Read all output from the process.
15     string metadata = process.StandardOutput.ReadToEnd();
16
17     // Wait for the process to exit.
18     process.WaitForExit();
19
20     // Terminate the process.
21     process.Kill();
22
23     return metadata;
24 }
```

In dieser Methode wird das Exiftool-Executable mithilfe der *Process*-Klasse aufgerufen. Der Aufruf, welcher in Zeile acht zusammengebaut wird, setzt sich aus der Option *-n* und dem Dateipfad zusammen. Das Argument *-n* ist in diesem Fall wichtig, damit alle numerischen Werte unformatiert und ohne Einheiten ausgegeben werden. Ohne dieses Argument würde die Dateigröße zum Beispiel mit „MB“ oder „GB“ abgekürzt werden, um es lesbarer zu machen, was die weitere Verarbeitung aber erschweren würde. Nach dem Prozessaufruf wird nur noch die Ausgabe gelesen und nach dem Beenden der Prozess wieder gestoppt und die Metadaten als String in folgendem Format zurückgegeben:

```
1  ExifTool Version Number         : 12.87
2  File Name                       : Dimplomarbeit_Panda.pdf
```

```
Directory                       : .
File Size                       : 584222
Zone Identifier                 : Exists
File Modification Date/Time     : 2025:02:03 13:36:45+01:00
File Access Date/Time           : 2025:03:30 11:20:32+02:00
File Creation Date/Time         : 2025:02:03 13:36:40+01:00
File Permissions                : 100666
File Type                       : PDF
File Type Extension             : PDF
MIME Type                       : application/pdf
...
```

Um nun diese Ausgabe in ein *Meta*-Objekt verwandeln zu können, müssen die relevanten Informationen extrahiert werden. Dazu wurde die wiederum private Methode *GetMetadataWithRegex* implementiert, welcher die gesamte Ausgabe des Exiftools und der Namen des Feldes, dessen Wert gesucht wird, übergeben wird.

```
private static string GetMetadataWithRegex(string input, string fieldName)
{
    // Build a regex pattern to find the metadata field followed by its value.
    string pattern = string.Format("(?<={0}\\s+:\\s)(.*)", fieldName);

    // Use regex to extract the field's value and trim any extra whitespace.
    return Regex.Match(input, pattern).Value.Trim();
}
```

Wie im Code Beispiel zu sehen, wird zuerst ein kompliziert aussehendes Regex-Pattern zusammengebaut, mit welchem später in der gesamten Ausgabe nach dem Wert zu dem jeweiligen Feld gesucht und mit *Trim* bereinigt wird. Zerlegt man dieses Regex-Pattern ist es gar nicht mehr schwer zu verstehen. Das Muster `(?<= .. )` stellt ein positives Lookbehind dar, dies bedeutet, es wird die Textstelle gesucht, die direkt nach dem darin beschriebenen Muster kommt. Im Lookbehind wird mit `{0}\s+:\s` der im Aufruf übergebene Wert (`{0} <= fieldName`), gefolgt von mindestens einem Leerzeichen (`\s+`), einem Doppelpunkt und einem weiteren Leerzeichen gesucht. Da dieser Teil im Lookbehind steht, wird dieser nicht zurückgeliefert, sondern das, was danach kommt. Dieser Teil kann in der sogenannten *Capture Group* weiter spezifiziert werden. Da wir bis zum Zeilenumbruch alles extrahieren wollen, sieht die Capture Group in diesem Fall so aus `(.*)`, was so viel heißt wie: Ein beliebiges Zeichen kann beliebig oft vorkommen.

Da nach dem Extrahieren der Werte mittels Regex immer ein String zurückgeliefert wird, ist es nötig, die Werte, die ein Datum oder die Dateigröße enthalten, auf den richtigen Datentypen zu parsen.

Um die Datum-Strings in ein DateTime-Objekt zu parsen, wurde die Methode *ParseDateFromString* implementiert:

```
1  private static DateTime? ParseDateFromString(string dateString)
2  {
3      // Define the expected date format.
4      string format = "yyyy:MM:dd HH:mm:ssK";
5
6      // Try to parse the date string with the exact format.
7      if (DateTimeOffset.TryParseExact(dateString, format,
       ↪  CultureInfo.InvariantCulture, DateTimeStyles.None, out DateTimeOffset
       ↪  dateTimeOffset))
8      {
9          // Return the DateTime component if parsing is successful.
10         return dateTimeOffset.DateTime;
11     }
12
13     // Return null if parsing fails.
14     return null;
15 }
```

In dieser Methode wird zuerst das Pattern definiert, in welchem das Datum im String steht; dieses ist beim Exiftool immer in einer Kombination aus sechs-Stellen für das Datum sowie die Zeit im 24-Stunden-Format inklusive der Zeitzoneninformation. Ein durch dieses Pattern beschriebenes Datum könnte zum Beispiel so aussehen: `2025:02:03 13:36:40+01:00`, dies entspricht genau dem Pattern: `yyyy:MM:dd HH:mm:ssK`.

In Zeile sieben der Methode wird mit der *DateTimeOffset* Klasse versucht, einen String nach dem gegebenen Muster zu parsen. Ist dies erfolgreich, kann ein Objekt der Klasse DateTime zurückgegeben werden. Andernfalls wird *null* retourniert. Die Übergabeparameter, die neben dem String, in dem das Datum gespeichert ist, und dem Muster, nach dem geparst werden soll, mitgegeben wurden, legen fest, dass ein kulturunabhängiges Format, zum Beispiel bei den Monatsnamen, verwendet wird und zudem keine zusätzlichen Anpassungen wie Zeitzonenkonvertierung oder Ähnliches vorgenommen werden sollen. Ist das parsen erfolgreich, wird das Ergebnis in die Variable `dateTimeOffset` der Klasse *DateTimeOffset* gespeichert.

Um die Dateigröße aus dem extrahierten String zu parsen, wurde die Methode *ParseFileSizeFromString* implementiert:

```
private static long? ParseFileSizeFromString(string sizeString)
{
    // Extract digits from the string using regular expression and parse it to long.
    long value = long.Parse(Regex.Match(sizeString, @"\d*").Value);
    return value;
}
```

In dieser Methode wird auch wieder ein regulärer Suchausdruck verwendet, damit wirklich nur Zahlen extrahiert werden. Das Ergebnis aus diesem Suchausdruck wird dann noch in den Datentyp long geparst und zurückgegeben.

Im ursprünglichen Aufruf von *GetMeta* wurden nun alle Werte extrahiert und geparst, sowie in einem Meta-Objekt zusammengeführt, welches nun zurückgegeben werden kann.

ContentExtractor

Die Klasse *ContentExtractor* wurde implementiert, um den Text aus den Dateien mithilfe von Tika zu extrahieren.

Da die ganze Magie des Textextrahierens im Tika-Service vonstattengeht, ist die Methode *GetText*, welche mit dem jeweiligen Dateipfad aufgerufen wird, nicht sonderlich kompliziert:

```
public async Task<string> GetText(string path)
{
    // Read all bytes of the file asynchronously.
    byte[] fileBytes = await File.ReadAllBytesAsync(path);
    // Create HTTP content from the byte array.
    var content = new ByteArrayContent(fileBytes);

    // Set the Accept header so that the server knows we expect plain text.
    _httpClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("text/plain"));

    // Send the file via a PUT request to the Tika API (endpoint "tika").
    var response = await _httpClient.PutAsync($"{Uri}tika", content);

```

```
14      // Check if the request was successful.
15      if (response.IsSuccessStatusCode)
16      {
17          Console.WriteLine("extracted successfully");
18          // Read and return the extracted text as a string.
19          return await response.Content.ReadAsStringAsync();
20      }
21
22      // If the request fails, throw an exception with an error message.
23      throw new Exception("Tika API request failed");
24  }
```

Damit die Datei übertragen werden kann, wird sie zuerst mithilfe der *File*-Klasse als ByteArray eingelesen und daraus ein HTTP-Content erstellt. In Zeile 9 wird außerdem noch der *MediaType*-Header auf `text/plain` gesetzt, damit sichergestellt wird, dass keine Datei, sondern reiner Text zurückgegeben wird.

Nach diesen Vorbereitungen wird dem HTTP-Client bei dem PUT-Request in Zeile 12 die Datei in Bytes übermittelt. War der Vorgang erfolgreich, kann der zurückgelieferte Content als String an den Aufrufer zurückgegeben werden. Ansonsten wird ein Fehler geworfen.

Chunker

Der letzte Schritt vor dem Indizieren der Dateien in die Datenbank ist das *chunken* der in den Dateien enthaltenen Contents, um die Größe der gespeicherten Dokumente zu verringern.

Der Methode *ChunkRecursive* wird der gesamte Content einer Datei übergeben um diesen rekursiv aufzuteilen und als Array von Strings zurückzugeben. Wie diese Art der Teilung funktioniert, wird im Kapitel LangChain genauer beschrieben.

```
1   public string[] ChunkRecursive(string content)
2   {
3       // Create a text splitter instance with the specified chunk size and overlap.
4       TextSplitter splitter =
5           new RecursiveCharacterTextSplitter(chunkSize: _chunkSize, chunkOverlap:
            ↪  _chunkOverlap);
6
7       // Split the text into chunks.
8       var chunks = splitter.SplitText(content);
```

```
9
10      // Output the number of generated chunks for debugging.
11      Console.WriteLine($"chunked successfully in {chunks.Count} chunks");
12
13      // Return the chunks as an array.
14      return chunks.ToArray();
15  }
```

Da das Aufteilen selber durch den *RecursiveCharacterTextSplitter* von *LangChain* durchgeführt wird, muss in dieser Methode nur die Konfiguration für diesen vorgenommen werden und damit die Methode *SplitText* aufgerufen werden. Da aus diesem Aufruf eine *IReadOnlyList* retourniert wird, muss diese vor dem Zurückgeben noch in ein Array umgewandelt werden.

Datenbankoperationen

Um die Datenbankoperationen unabhängig zu managen, wurde das Interface *IDBClient* implementiert. In diesem Interface werden alle Methoden definiert, die zum Einfügen und Updaten der Datenbank notwendig sein werden, implementiert. Das Interface enthält demnach folgende Methoden:

```
1  void AddEntry(Document document);
2  void DeleteEntries(string fileName);
3  Task<List<Document>> SearchDocumentsByFileNameAsync(string fileName);
4  Task<List<Document>> GetAllIndexedDocumentsAsync();
```

Elasticsearch

Um die notwendigen Methoden im Elasticsearch Client zu implementieren, muss beim Instanziieren der Klasse ein Objekt der Klasse *ElasticsearchClient* mit der Elasticsearch-URL, dem dazu passenden API-Key sowie dem Index-Namen erstellt werden.

Das Hinzufügen eines neuen Dokuments ist relativ unspektakulär, da der Client die nötige asynchrone Methode bereitstellt. Es muss nur angegeben, welches Dokument in welchen Index indiziert werden soll.

```
1  public async void AddEntry(Document document)
2  {
3      var response = await client.IndexAsync(document, i => i.Index(IndexName));
4
```

```
5     Console.WriteLine(response.IsSuccess() ? "Indexed successfully" : "Indexing not
      ↪  successful");
6 }
```

Auch das Löschen eines bestimmten Dokuments anhand des Dateinamens ist nicht sehr kompliziert.

```
1 public async void DeleteEntries(string fileName)
2 {
3     // Delete documents in Elasticsearch based on the file name.
4     var response = await client.DeleteByQueryAsync<Document>(d => d
5     .Query(q => q.Term(t => t.Field("meta.fileName.keyword").Value(fileName))));
6     Console.WriteLine(response.IsSuccess() ? "Deleted old entries" : "Failed to
      ↪  delete old entries");
7 }
```

In Zeile 4 und 5 werden nur die Dokumente gelöscht, welche im Feld fileName unter Meta genau den übergebenen Wert enthalten.

Relativ ähnlich sieht es aus, wenn man in einem Index nach allen Dokumenten mit einem bestimmten Dateinamen sucht:

```
1  public async Task<List<Document>> SearchDocumentsByFileNameAsync(string fileName)
2  {
3      var response = await client.SearchAsync<Document>(s => s
4          .Index(IndexName)
5          .Query(q => q.Term(t => t.Field("meta.fileName.keyword").Value(fileName)))
6          .Size(1000)
7      );
8
9      var docs = new List<Document>();
10     if (response.IsValidResponse && response.Hits != null)
11     {
12         foreach (var hit in response.Hits)
13         {
14             docs.Add(hit.Source);
15         }
16     }
17     return docs;
18 }
```

Hier wird wieder nach allen Dokumenten gesucht, welche genau den übergebenen Wert im Feld fileName gespeichert haben. Standardmäßig gibt Elasticsearch immer nur die ersten 10 Hits zurück, daher muss explizit angegeben werden, wie viele Dokumente zurückgegeben werden sollen. Weil nicht damit zu rechnen ist, dass ein Dokument mehr als 1.000 Hits liefert, wurde die Size in diesem Fall auf 1.000 gesetzt. Da diese Hits als *SearchResponse* Objekt gespeichert werden, werden sie anschließend noch in eine Liste von *Document* Objekten überführt.

Um möglichst alle Dokumente zu erhalten, wird beim nächsten Aufruf die Size auf den Maximalwert von 10.000 gesetzt. Wäre zu erwarten, dass mehr als 10.000 Dokumente abgefragt werden müssen, müsste eine andere Art der Abfrage gewählt werden, wie zum Beispiel die Elasticsearch Scroll-API.

```
1  public async Task<List<Document>> GetAllIndexedDocumentsAsync()
2  {
3      // Execute a MatchAll query to retrieve all documents.
4      var response = await client.SearchAsync<Document>(s => s
5          .Index(IndexName)
6          .Query(q => q.MatchAll(new
           ↪  Elastic.Clients.Elasticsearch.QueryDsl.MatchAllQuery()))
7          .Size(10000)
8      );
9
10     var docs = new List<Document>();
11     if (response.IsValidResponse && response.Hits != null)
12     {
13         foreach (var hit in response.Hits)
14         {
15             docs.Add(hit.Source);
16         }
17     }
18     return docs;
19 }
```

Bei diesem Aufruf wird mit dem *MatchAllQuery* nach allen verfügbaren Dokumenten im Index gesucht. Der Rest ist ähnlich zu dem der Suche nach einem bestimmten Dateinamen.

Milvus

Die Speicherung der Daten in Milvus wird auch durch das Programm **DocumentDissector** umgesetzt. Dabei werden Dokumente in kleinere Abschnitte unterteilt, mit einem LLM in Vektoren umgewandelt und anschließend in der Vektor-Datenbank Milvus gespeichert.

Die konkrete Speicherung erfolgt über die Klasse `MilvusClient`, welche beim Initialisieren alle notwendigen Konfigurationen ausliest, eine Verbindung zur Milvus-Instanz herstellt und die Daten über eine HTTP-API einfügt. Dabei wird geprüft, ob die Ziel-Collection bereits existiert, falls nicht, wird sie automatisch erstellt.

Für eine detaillierte Beschreibung der Installation, Konfiguration und der einzelnen Methoden siehe Kapitel 4.3.2.

4.5.3 Businesslogik

Projektaufbau

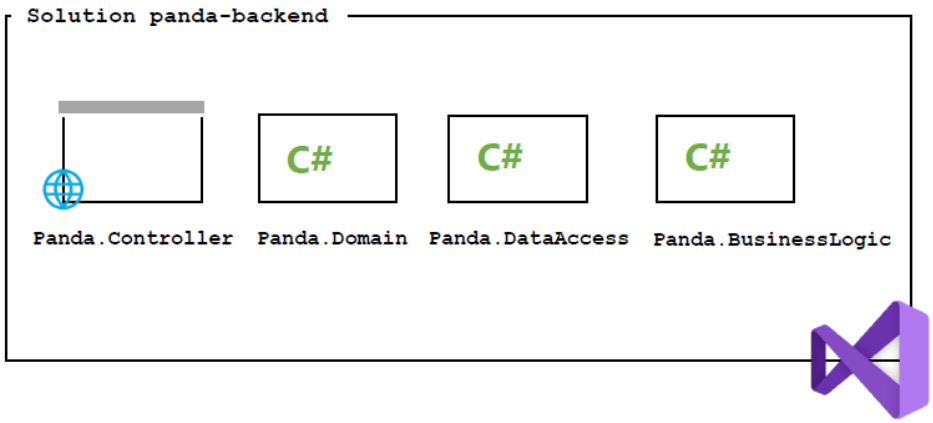


Abbildung 45: Projektmappe Backend

Die Businesslogik enthält im Paket **Panda.BusinessLogic** den AnsweringService für die Beantwortung einer Frage und den ChatService für die Verwaltung der Chats im Frontend. Die für beide Services modellierten Klassen sind im Paket **Panda.Domain** beschrieben. Die Migrationen der Chat-Datenbank, der DB-Kontext sowie weitere Datenbank-spezifische Konfigurationen befinden sich in dem Paket **Panda.DataAccess**.

Die REST-API **Panda.Controller**, genauer beschrieben in Kapitel 4.2, ist die Konsolenanwendung, die Endpunkte hostet und die Prozesse durchführt. Sie ist mit dem ASP.NET Web Application-Template aufgebaut worden. In ihrem Programmablauf werden die Methoden und Klassenbeschreibungen der anderen Pakete verwendet.

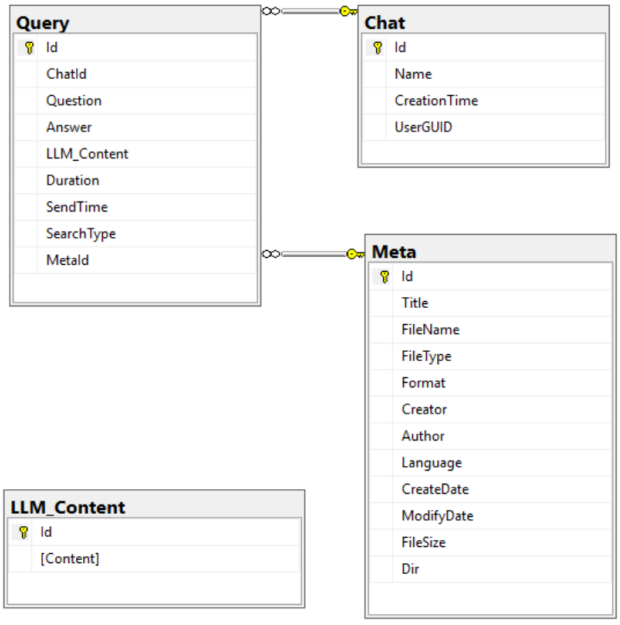


Abbildung 46: ERD-Diagramm der Entitäten im Backend

Entitäten

In der Verwaltungs-Datenbank werden **Chats** gespeichert, welche über eine UserGuid eindeutig einem Benutzer zugeordnet sind. Jeder Chat enthält mehrere **QueryItems**. Dieses Objekt speichert die Frage und Antwort eines Befragungsprozesses. Jedes QueryItem verweist auf ein **Meta-Objekt**. Diese enthält Metadaten des zugehörigen Dokuments. Zusätzlich gibt es noch das **LLM_Content-Objekt**, welches im Frontend gewählt und beim QueryItem direkt als String eingetragen werden kann. Dieser Text wird bei der Dokumentenabfrage hinzugefügt, um nach spezifischeren Antworten zu suchen (genauere Beschreibung siehe Abschnitt 4.3.3).

4.5.4 Frageantwortprozess

Der Frageantwortprozess bildet den **Hauptprozess** des RAG-Systems. In diesem wird eine Frage gestellt, danach verarbeitet und mithilfe von Dokumenten beantwortet.

Die Beantwortung der Frage übernimmt der **AnsweringService**. Dieser Service benötigt zur Initialisierung die **Verbindungsoptionen** der Milvus- und Elasticsearch-Datenbank, der OpenAI-API, des SSO-Client sowie der SQL-Server-Verwaltungsdatenbank. Diese müssen in den **appsettings.json** eingetragen sein und werden über einen Selektor ausgelesen. Das Dokument bietet eine einfache Übersicht und Bearbeitung, um die Verbindungsoptionen der Komponenten austauschbar zu halten.

Der AnsweringService prüft als Nächstes, ob der lokale- oder online-Prozessablauf gewählt worden ist und deklariert eine LLM-Entität . Dies ist auch eingetragen in den appsettings.json. Je nach eingetragener Art wird eine neue Ollama- oder OpenAI-LLM-Entität erstellt. In den Klassen sind Methoden implementiert, die mittels Http-Requests die **Generierung eines Suchvektors** und die **Generierung eines Elasticsearch-Suchquery** durchführen.

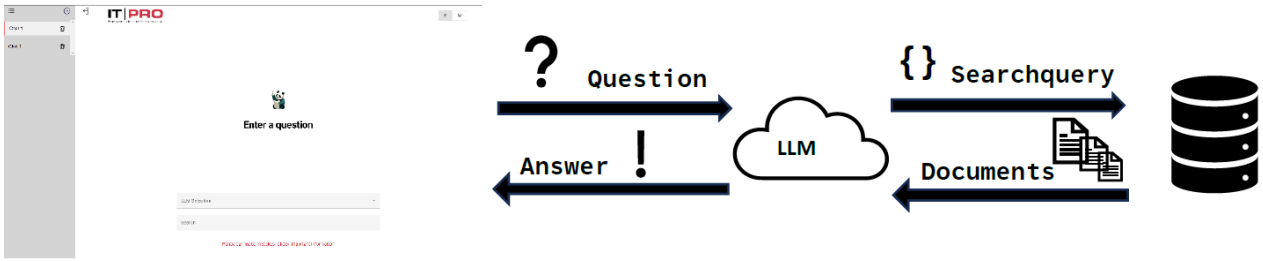


Abbildung 47: Frageantwortprozess

Generell fängt der Frageantwortprozess im Frontend an. Ein Benutzer gibt eine Frage in einem Chat ein. Das Frontend sendet die Frage weiter an das Backend. Nach der Authentifizierung und Autorisierung wird sie an das AnsweringService weitergeleitet. Dieser Service initialisiert, wie zuvor beschrieben, eine **LLM-Entität**. In diesem wird mithilfe von OpenAI eine **Suchanfrage** (Searchquery) erstellt. Mit dieser Query wird in der, im Frontend gewählten, **Suchdatenbank** nach Dokumenten **gesucht**. Dabei ist zwischen der Milvus und Elasticsearch-Datenbank zu unterscheiden. Bei ersterem beinhaltet der Searchquery einen **Vektor**, bei zweitem eine **Elasticsearch-Abfrage** (siehe 4.5.4). Die Datenbank gibt dann eine **Liste** von gefundenen **Dokumenten zurück**. Das am meisten Zutreffende ist das erste Dokument, dieses wird ausgewählt. Um die ursprüngliche Frage korrekt und in der richtigen Sprache zu beantworten wird der Inhalt in einem Prompt an das LLM geschickt und eine Antwort in **natürlicher Sprache** gefordert. Die **Antwort** sowie die **Metadaten** des ursprünglichen Dokuments werden gemeinsam als fertiges **QueryItem** an das Frontend zurückgegeben, wo der Benutzer seine Antwort und Dateidaten zu sehen bekommt.

Mögliche Optionen

Das RAG-System hat zwei Komponenten, von denen jeweils zwei Optionen zur Verfügung stehen:

1. Dokumenten-Datenbank: Milvus und Elasticsearch

   Diese Datenbanken enthalten indizierte Dokumente. Beide Datenbanken unseres Projekts sind für eine effiziente, schnelle Suche spezialisiert. Ihr Unterschied befindet sich

im Algorithmus. Milvus speichert Texte als Vektoren ab. Sie verwendet eine semantische Suche um Eingangsvektoren mit den Gespeicherten zu vergleichen (siehe 2.3.2). Elasticsearch speichert die Texte als Dokumente in einem Index. Die Datenbank verwendet eine Volltextsuche um Vergleiche des Eingangstexts mit den Gespeicherten zu machen (siehe 2.3.1).

Die Auswahl der Suchdatenbank erfolgt im Frontend. Weitere Datenbanken können im Backend hinzugefügt werden, es stellt aber einiges an Aufwand dar. Es muss eine **Konfigurationsklasse** für die Datenbank geschrieben, eine **Prozessmethode im AnsweringService** korrekt hinzugefügt und die **Verbindungsoptionen** in den appsettings.json eingetragen werden.

2. Large Language Model: OpenAI und Ollama

   Das Large Language Model ist die wichtigste Hilfestellung, da es Wörter für die Suchanfrage und die finale Antwort generiert. In diesem Projekt wird hauptsächlich die **OpenAI-API** verwendet. Das LLM ist online ausgelagert und liefert Antworten mit der besten Qualität. Die Anfragen werden mit einem API-Key gestellt, dabei entstehen Kosten. Als weitere LLMs wurden **llama2** für die Promptabfrage und **all-minilm** für die Vektorgenerierung verwendet. Der lokal gehostete Ollama-Server stellt die Prozesse eines LLM kostenfrei zur Verfügung, dabei entsteht aber ein hoher Prozessorbedarf. Der Server wurde getestet und analysiert. Da die Qualität der Antworten nicht befriedigend ist und eine Abfrage mehrere Minuten dauert, ist diese Option wieder verworfen worden.

In folgendem Code-Beispiel wird ein LLM-Objekt gemäß der ausgewählten Optionen gesetzt sowie der Beantwortungs-Prozess lexikalisch oder semantisch durchgeführt. Mit dem **configuration**-Objekt werden die Einstellungen der appsettings.json an das LLM-Objekt übergeben.

```
public async Task<Document> ExecuteQueryAsync(Query query)
{
    ILLM llm;
    if (configuration.GetValue<bool>("GlobalOptions:Run:Local"))
    {

        llm = new OllamaLLM(new OllamaConfig(
            configuration.GetConnectionString("OllamaURI"),
            configuration.GetValue<string>("GlobalOptions:Ollama:LLM_Model"),
            configuration.GetValue<string>("GlobalOptions:Ollama:Embedding_Model")
            ));
    }
    else
    {
        llm = new OpenAILLM(new OpenAIConfig(
            configuration["ConnectionStrings:OpenAIURI"],
            configuration["GlobalOptions:OpenAI:Key"],
            configuration["AppConfig:OpenAI:DefaultModel"],
            configuration["AppConfig:OpenAI:DefaultRole"]
            ));
    }

    if (query.SearchType == Domain.SearchType.Lexical)
    {
        return await ExecuteLexicalAsync(llm, query);
    }
    if (query.SearchType == Domain.SearchType.Vector)
    {
        return await ExecuteVectorAsync(llm, query);
    }
    throw new InvalidOperationException("SearchType not valid");
}
```

Abbildung 48: Methode zur Auswahl des LLM, Konfigurationen werden über die appsettings geladen

Beantwortung Ablauf

Für die Beantwortung wird das korrekte Dokument in der Suchdatenbank benötigt, da der Benutzer im Regelfall nach dem Inhalt genau eines Dokuments sucht. Die Suchanfrage zur Datenbank wird mit vordefinierten Templates und Daten von OpenAI erstellt.

Die Datenbank-Konfigurations-Klasse ist eine Sammlung an **Prompts** und **Abfragevorlagen**. Mittels eines Prompts werden Daten von OpenAI angefragt. Dann wird eine Abfrage an die Suchdatenbank gebildet. Im Projekt sind auch zwei verschiedene Arten für den Bau der Abfrage in Elasticsearch getestet worden, diese sind im nächsten Kapitel 4.5.4 beschrieben.

Die Suchdatenbanken geben einen bestimmten Abschnitt eines Dokuments bei einer Suche zurück.Dieser **Inhalt** des Dokuments und die ursprüngliche **Frage** wird wieder in einen Prompt

eingebaut und an OpenAI versendet. Das LLM erstellt eine natürliche Antwort, welche in den Query eingebaut werden kann und an den Benutzer gesendet wird.

```
1  public async Task<Document> ExecuteLexicalAsync(ILLM llm, Query query)
2   {
3       ElasticsearchClient client = new(elasticsearchConfig.GetConnectionSettings());
4
5  // Wörter von OpenAI zum Bau der Suchabfrage
6  string llmResponse = await
   ↪  llm.GetLLMResponseAsync(elasticsearchConfig.GetOpenAIQuerySimple(query.Question,
   ↪  query.LLM_Content));
7
8  // Einbau der Wörter in eine Abfragevorlage
9  string elasticsearchQuery = elasticsearchConfig.GetQuery1(llmResponse);
10
11 // Anfrage an ES mittels Suchabfrage
12 Document doc = await GetElasticsearchResponseAsync(client, elasticsearchQuery);
13
14 // Anfrage an OpenAI für die natürliche Antwort
15 string answer = await
   ↪  llm.GetLLMResponseAsync(elasticsearchConfig.GetOpenAICombine(query.Question,
   ↪  doc.Content));
16
17 return doc;
18 }
```

Abbildung 49: Frageantwortprozess im Elasticsearch-Ablauf

Der Ablauf mit der Milvus-Datenbank unterscheidet sich von Obigem darin, dass aus der Suchanfrage (Zeile 6) ein Vektor generiert wird. Dieser wird dann für die Suche in der Milvus-Datenbank verwendet.

Bau der Abfrage in Elasticsearch

Für den Bau der Abfrage sind in Elasticsearch zwei verschiedene Arten entwickelt worden. Im Milvus Datenbank-Abfrageprozess gibt es nur eine Art.

1. OpenAI übernimmt gesamten Zusammenbau

   Im ersten Fall wird ein Prompt an OpenAI geschickt und der zurückgegebene Wert ist eine JSON-Abfrage, welche direkt an Elasticsearch geschickt werden kann. Der Prompt enthält eine **genaue Bauanleitung**. Wichtig ist, **welche Wörter** gesucht sind, **wo** sie eingefügt gehören und die **Form** des Rückgabewerts. Um in diese Abfrage keine Fehler einzubauen und ihn qualitativ hochwertig zu gestalten werden **Prompting Techniken** (siehe 4.5.4) verwendet.

```
        public string GetOpenAIQuery(string question, string llmContent)
    {
        string defaultLLMContent = $@"Nimm aus der Frage alle Nomen sowie viele relevanten Worte (außer Stoppworte) und füge selbst viele ähnliche Wörter hinzu. Die Sprache der Wörter muss gleich bleiben.
                                    Füge diese Wörter dem Query mit den richtigen Feldern (Beispielquery) hinzu

                                    Frage: {question}
                                    Hilfreicher Text:";
        if (llmContent.IsNullOrEmpty()) return optimizedQuery1 + defaultLLMContent + toJsonFormatString + "\n";
        return optimizedQuery2 + llmContent + toJsonFormatString + "Frage:"+question;
    }
```

Abbildung 50: OpenAI generiert gesamten Query

2. OpenAI erstellt Suchwörter für Abfragevorlage Im zweiten getesteten Ablauf übernimmt OpenAI **nur** die **Generierung von Wörtern**. Dabei wird die ursprüngliche Frage an das LLM gesendet und die Schlüsselwörter werden extrahiert. Die Elasticsearch-**Suchabfragen** liegen als **Vorlage** vor, diese sind von uns ausgearbeitet worden. Da wir als **Diplomarbeitsteam** das Format im Index vorgegeben haben, können die genauen Felder angegeben werden, welche abgefragt werden sollen. Der Inhalt dieser Felder wird mit den Wörtern von OpenAI gefüllt.

```
public string GetOpenAIQuerySimple(string question, string llmContent)
    {
        string defaultLLMContent = $"gib mir die schlüsselwörter aus folgender frage in reinster form";

        if (llmContent.IsNullOrEmpty())
        {
            return defaultLLMContent + "\n question: " + question;
        }
        else
        {
            return defaultLLMContent + "(zusatzinfos: " + llmContent + ")\n question: " + question;
        }
    }
```

Abbildung 51: OpenAI generiert nur signifikante Wörter

Elasticsearch-Suchanfragen generieren

Es gibt verschiedene Abfrage-Algorithmen in Elasticsearch und jeder funktioniert in bestimmten Fällen besser. Die folgenden Parameter müssen berücksichtigt werden, um den optimalen Abfrage-Typen zu finden:

- Gespeicherte Dokumente: Die gespeicherten Dokumente in unserem Projekt sind **sehr lang**. Die Anzahl der Zeichen wird **ChunkSize** genannt und beträgt fast die Länge einer maximalen OpenAI-Anfrage. Bei bestimmten Anfragen haben lange Dokumente eine **größere Trefferquote**, da **Sonderzeichen** wie \n berücksichtigt worden sind. Das muss verhindert werden.
- Eingangsparameter: Stellt ein Benutzer eine Frage, so hat er eventuell **keine Ahnung vom Aufbau** des gespeicherten Dokuments. Von der Frage schneidet OpenAI die **signifikanten Wörter** heraus. Mit diesem Schritt werden Wörter mit wenig Bedeutung (Wie, und, auch...) nicht mehr berücksichtigt. Der Nachteil ist, dass die **Reihenfolge der Wörter** nach der Beseitigung irrelevanter Wörter **keinen Sinn** mehr macht. Dies muss auch berücksichtigt werden.

Im Laufe des Projekts wurden die Abfragetypen match, multi_match, bool und match_phrase getestet. Die Funktionsweise der Typen sind im Kapitel 4.3.1 „Elasticsearch“ beschrieben.

Der **Abfragetyp** wird **manuell** bestimmt. Wie im vorigen Kapitel beschrieben wird eine Abfragevorlage mit dem Abfragetypen direkt in einem Prompt eingebaut oder nur die signifikanten Wörter eingefügt. Eine Beispiel-Suchanfrage ist in Abschnitt 4.5.4, Abbildung 54 zu sehen.

Milvus-Suchanfragen generieren

Die Suchanfragen an Milvus werden ebenfalls von OpenAI mithilfe von vordefinierten Prompts generiert. Der Unterschied liegt in den **Vector-Embeddings**. Statt einer Query, die die wichtigsten Wörter beinhaltet, sucht die Milvus-Datenbank mit einem Vektor, repräsentiert in einer **langen Liste aus Gleitkommazahlen**.

Der **Frageanwortprozess** bleibt identisch, nur wird dazwischen eine weitere Methode aufgerufen, um den Vektor mithilfe von **OpenAI** zu **generieren**. Dieser **Vektor** wird in ein JSON-Objekt eingebaut, um eine Datenbank-Abfrage durchzuführen. Diese Datenbank sendet Dokumente zurück, welche, auf **demselben Weg** wie bei Elasticsearch, für eine korrekte Antwort **weiterverarbeitet** werden.

```
{"collectionName":"documents","data":[[0.018055383,-0.0054087476,
                                       ...,
                                       0.0032534436,-0.008804786]],
"annsField":"vector","limit":3,"
outputFields":["id","vector","document"]
}
```

Abbildung 52: Searchquery, mit welchem Daten von **Milvus** abgefragt werden

Durchführung eines Beispielszenario

In folgendem Abschnitt ist ein Beispielszenario dargestellt. Ein autorisierter Benutzer sendet von einem Chat aus eine Frage an das Backend, als Suchdtenbank ist Elasticsearch ausgewählt. Das Szenario wird mit beiden Arten des Abfrage-Baus durchgespielt.

Frontend

```
Wie viele Kosten können im Total Cost of Ownership Dokument von Azure
eingespart werden?
```

Abbildung 53: Frage eines Benutzers

Die ursprüngliche Frage **ändert sich** für die Suchanfrage **signifikant**. Übernimmt OpenAI die Bauanleitung wird die Reihenfolge nicht mehr berücksichtigt, Bindewörter verworfen und wichtige Wörter öfter und mit Synonymen vervielfältigt. Diese Frage wird mit Prompting-Techniken bearbeitet, um eine gezielte Suchanfrage generieren zu lassen. Liegt bei der zurückgegebenen Query keine syntaktische Korrektheit vor, so wirft Elasticsearch einen Fehler. Dies gilt es zu vermeiden.

Wird bei der zweiten Art die Vorlage als Abfrageprozess verwendet, werden nur die Schlüsselwörter von OpenAI zurückgegeben.

```
{
    "query": {
        "multi_match": {
            "query": "1001",
            "fields": ["meta.title", "content", "meta.fileName"],
            "type": "most_fields"
        }
    }
}
Nimm aus der Frage alle Nomen sowie viele relevanten Worte (außer Stoppworte) und
füge selbst viele ähnliche Wörter hinzu. Die Sprache der Wörter muss
gleich bleiben.
Füge diese Wörter dem Query mit den richtigen Feldern (Beispielquery) hinzu
Frage: Wie viele Kosten können im Total Cost of Ownership Dokument von Azure
eingespart werden?
Hilfreicher Text:lösche 1001 im json. Füge die einzelnen ausgearbeiteten Wörter
wiederholt ein und gib das Query unbedingt im richtigen Json-Format zurück
(meta.title,meta.filename,content)
```

Abbildung 54: Prompt für die Erstellung einer Suchanfrage (Bauanleitung)

```
gib mir die schlüsselwörter aus folgender frage in reinster form
 question: Wie viele Kosten können im Total Cost of Ownership
 Dokument von Azure eingespart werden?
```

Abbildung 55: Prompt für die Erstellung von signifikanten Wörtern (Vorlage)

Der erste Prompt wird von OpenAI verarbeitet und auf ein JSON-Objekt gewartet. Die Antwort, die **bei jeder Anfrage** ein JSON-Objekt sein muss, wird zurückgegeben. Das wird sichergestellt, indem im Prompt direkt nach einem JSON-Objekt gefragt wird und auch eine Beispielanfrage mitgesendet wird. Das RAG-System teilt OpenAI mit, dass die wichtigsten Informationen in der Frage vorhanden sind. Der Prompt stellt auch sicher, dass die Suchanfrage mit der Eingangssprache übereinstimmen soll (weitere Informationen zum Prompting in Abschnitt 4.5.4.

Der zweite Prompt gibt in einem String die wichtigsten Wörter zurück. Diese werden in die Vorlage eingebaut und dann an Elasticsearch gesendet.

```
{
    "query": {
        "multi_match": {
            "query": "Kosten sparen einsparen verringern reduzieren Total
                      Cost of Ownership Dokument Azure",
            "fields": ["meta.title", "content", "meta.fileName"],
            "type": "most_fields"
        }
    }
}
```

Abbildung 56: Suchanfrage an Elasticsearch (Bauanleitung)

```
{
  "query": {
    "multi_match": {
      "query": "Kosten, Total Cost of Ownership, Dokument, Azure,
      eingespart, werden.",
      "fields": [
        "content^3",
        "meta.title^2",
        "meta.author"
      ],
      "type": "best_fields"
    }
  }
}
```

Abbildung 57: Suchanfrage an Elasticsearch (Vorlage)

Diese JSON-Objekte sind die finalen Suchanfragen. In den Einstellungen des RAG-Systems ist ein bestimmter Elasticsearch-Index definiert. Dieser wird mittels der erstellten **Suchanfrage (multi_match)** durchsucht.

Elasticsearch baut intern eine boolsche-Abfrage auf. Diese ist nach außen hin nicht sichtbar. Das RAG-System erhält dann eine Liste von Dokumenten zurück. Der **Inhalt** des **ersten Dokuments** wird gemeinsam mit der ursprünglichen Frage an die OpenAI-API gesendet. Das RAG-System erhält die finale Antwort, welche an den Benutzer versendet wird.

Frontend

```
Die Kosten, die im Total Cost of Ownership Dokument von Azure eingespart
werden können, belaufen sich auf insgesamt €502,821.
```

Abbildung 58: Antwort an den Benutzer (Bauanleitung

Frontend

```
Insgesamt können im Dokument des Total Cost of Ownership von Azure Kosten
in Höhe von bis zu €502,821 eingespart werden.
```

Abbildung 59: Antwort an den Benutzer (Bauanleitung

Prompting Technik

Prompts im Allgemeinen sind vom Menschen geschriebene Anweisungen für KI-Systeme. Sie sind die **Kommunikationsschnittstelle** und führen zu einer Aktion oder bestimmten Antwort. Ein Prompt kann Zeichen, Bilder, Audiosignale und Weiteres enthalten. Je nach KI-System werden die Ressourcen anders verarbeitet und eine Antwort gebaut. [46]

Diese Prompts sind **das Schlüsselelement**, um Informationen aus einem KI-System zu extrahieren. Die Qualität einer Antwort hängt maßgeblich davon ab. In der Arbeitswelt rund um künstliche Intelligenz gibt es deswegen auch den Beruf des Prompt Engineers.

In unserem RAG-System sind einige Prompts ausprobiert worden. Der in Abbildung 54 gegebene Prompt führt mit hoher Wahrscheinlichkeit zur richtigen Antwort. Aufgrund der Forderung nach Synonymen und Zusatzwörtern kann man auch von der **Query-Expansion-Technik** [47] reden. Da dem Benutzer der Anwendung nicht bewusst ist, in welcher Form das Gesuchte in den indizierten Dokumenten steht kann der Prompt um einige Wortabwandlungen erweitert werden. So wird später lexikalisch und mit Vektoren **auch nach ähnlichen Wörtern gesucht**, was mit einer **höheren Wahrscheinlichkeit** zum richtigen Dokument führt.

Folgende **Elemente im Prompt** helfen der KI einen passenden Searchquery, verpackt in einem syntaktisch korrekten JSON-Objekt, zurückzugeben.

- Anleitung

Die KI braucht einen Hauptteil, um bei jedem Abruf dasselbe durchzuführen. Der Prompt in Abbildung 54 gibt dem LLM vor, alle Nomen und relevante Worte außer den Stoppworten aus der Frage zu filtern und ähnliche Wörter hinzuzufügen. Diese sollen dem **Beispielquery** hinzugefügt werden, welcher als **Ergebnis** zurückgegeben werden muss.

- Payload

Die ursprüngliche Frage bildet den sogenannten Payload. Es ist die Ressource, mit welcher die KI ähnliche Wörter aus ihrem System heraussuchen soll und in den Beispielquery einfügt.

- Kontext

Das sind Hilfsanweisungen im Prompt, um die Antwort der KI einzugrenzen. Beim Prompt in Abbildung 54 sind das die Aufforderung nach der gleichen Ein-/Ausgangssprache, die Löschung des Beispieltext im JSON-Objekt und die Angabe der meta-Felder, welche das LLM mit Wörtern abfragen soll.

- Beispiel

Ein LLM gibt viel genauere Ergebnisse zurück wenn ein Beispiel gegeben ist. Das System weiß dann, in welcher Form das Ergebnis zurückgegeben werden soll und wird jedes Mal diese Form einhalten. Im Beispiel-Prompt in Abbildung 54 ist das Beispiel der Query, welcher den Abfrage-Algorithmus, den Typen und die Felder schon vorgegeben hat. Die Zeichenkette „1001“ wird mit den Wörtern des Payload ersetzt.

4.6 Analyse

4.6.1 Analyzer

Der in Python geschriebene Analyzer ist ein speziell entwickeltes Dienstprogramm, das das Testen des Frage-Antwort-Prozesses unterstützt und die Ergebnisse im CSV-Format speichert und auswertet.

Analysevorbereitung und Datenbeschaffung

Das in Python geschriebene GUI Programm ermöglicht die Auswahl einer Frage, mit der gestartet werden soll sowie des LLM-Contents welcher verwendet werden soll. Außerdem können weitere relevante Attribute wie die Indizierungsmethode oder die im Backend eingestellte Abfragemethode für Elasticsearch eingestellt werden.

Die gesamte Konfiguration passiert in der TKinter GUI. Wie in Abbildung 60 zu sehen, werden bestimmte Einstellung vor dem Programmstart definiert.

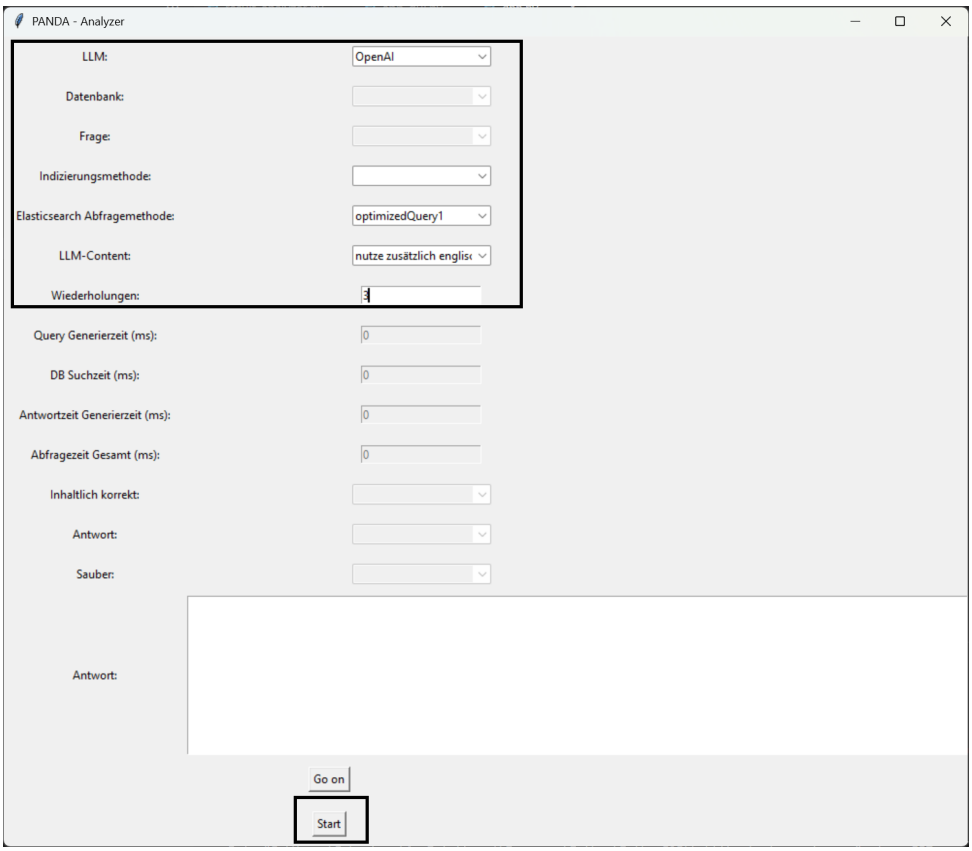


Abbildung 60: Konfigurationen vor dem Beginnen

Nachdem die vor Programmstart benötigten Informationen eingefügt wurden, kann mit dem Drücken des Start-Buttons der Analysevorgang beginnen.

Beim Start des Analysevorgangs wird der Request im API Client zusammengebaut und kann dann folgendermaßen aussehen:

```
{
    "id": 0,
    "chatId": 0,
    "duration": 0,
    "sendTime": "2025-03-05T08:18:43.832Z",
    "question": "Wie viele Kosten können im Total Cost of Ownership Dokument von
    ↪ Azure eingespart werden?",
    "searchType": 1,    // 1 .. Elasticsearch, 2 .. Milvus
    "answer": "string",
    "llM_Content": "nutze zusätzlich englische fachbegriffe und synonyme",
    "meta": {
        "id": "0",
        "title": "",
        "fileName": "",
        "fileType": "",
        "format": "",
        "creator": "",
        "author": "",
        "language": "",
        "createDate": "0001-01-01T00:00:00Z",
        "modifyDate": "0001-01-01T00:00:00Z",
        "fileSize": "0",
        "dir": ""
    }
}
```

Die meisten Attribute wie die IDs oder Metadaten sind bei den Requests, vor allem beim Analysieren irrelevant, müssen aber mitgesendet werden. Die wichtigsten Felder sind in diesem Fall die Frage (question), der SearchType, welcher bestimmt auf welche Datenbank abgefragt werden soll, dabei steht 1 für Elasticsearch und 2 für Milvus, sowie das Feld llM_Content, welcher, je nach Konfiguration im Vorhinein, leer sein darf oder mit dem ausgewählten LLM-Content befüllt ist.

Nachdem dieser Request abgearbeitet wurde, werden noch die Zeiten der einzelnen Schritte im Backend über den Analyse-Endpunkt des Backends geladen.

Nachdem die insgesamt fünf Requests erfolgreich abgearbeitet wurden, werden die Ergebnisse, wie in Abbildung 61 zu sehen, im Frontend angezeigt und es kann mit der Bewertung der Antwort begonnen werden.

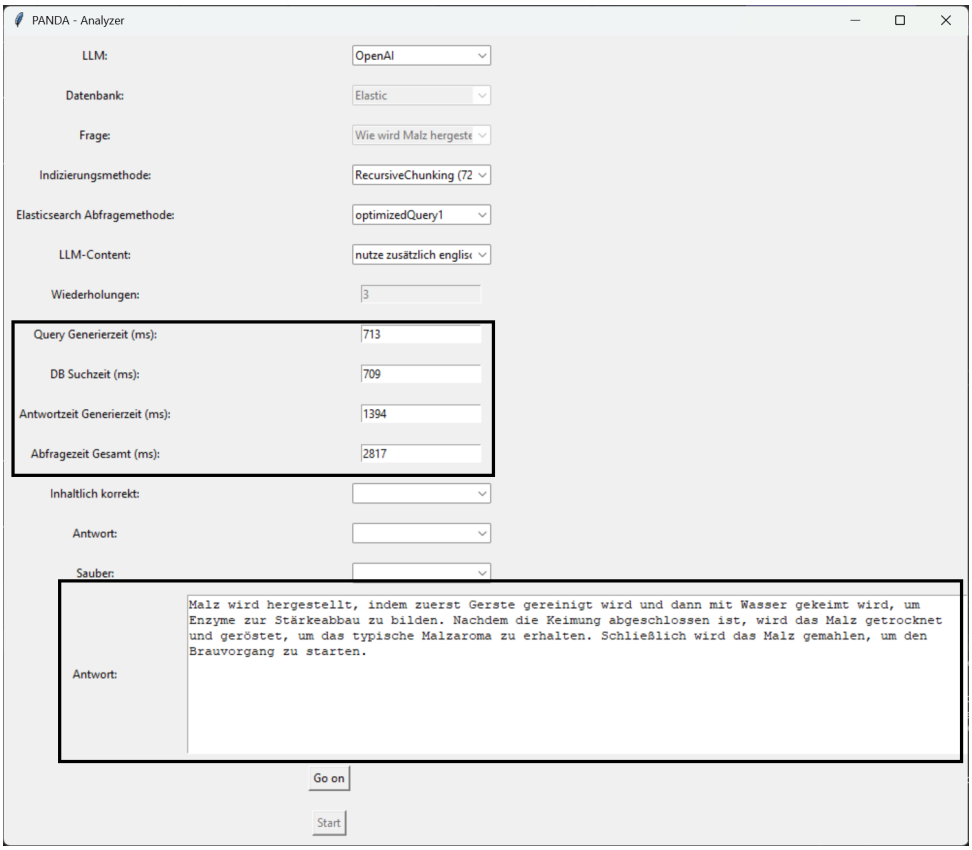


Abbildung 61: Analyzer nach den Requests

Die Bewertung wurde zu Beginn des Projekts auf drei einzelne Punkte aufgeteilt:

- **Inhaltlich korrekt:** Dieses Feld ist das Wichtigste dieser drei sowie der gesamten Analyse. Liefert der Response genau die Antwort, die erwartet wird zurück, wird der Wert auf „JA“ gesetzt. Wurde nur ein Teil der Frage in der Antwort beantwortet, ist der Wert auf „TEILWEISE“ zu setzen. Wenn auf die Frage zwar eine Antwort erstellt wird, diese aber nicht korrekt ist, wird dieser Wert auf „NEIN“ gesetzt.
- **Antwort:** Das Feld „Antwort“ beschreibt im Endeffekt, ob die generierte Antwort auch wirklich eine Antwort auf die gestellt Frage liefert oder ob sie gemeinsam mit der Frage keinen Sinn ergibt.
- **Sauber:** Zu Beginn des Projekts gab es oft das Problem, dass bei der Generierung der natürlichen Antwort durch das LLM „Antwort:“, „:“ oder ähnliches vor der eigentlichen Antwort geschrieben wurde. Da dieses Verhalten nicht gewollt war und für unseren Zweck

unschön ist, wurde das Attribut „Sauber“ erstellt, welches den Zweck hat so die Qualität der Antwort zu beschreiben.

Für alle drei Felder gibt es die Optionen „JA“, „NEIN“, „TEILWEISE“.

Nachdem diese drei Felder ausgefüllt wurden, kann mit dem Button „Go on“ fortgefahren werden. Alle Daten werden dann in einer CSV-Datei für spätere Auswertungen gespeichert. Je nach Konfiguration wird dann die Frage noch einmal gestellt, zur jeweils anderen Datenbank gewechselt oder zur nächsten Frage weitergegangen.

Analyse Auswertung

Nachdem mit dem Hilfsprogramm einige Requests durchgeführt und die Ergebnisse in der CSV-Datei gespeichert wurden, können diese verarbeitet und ausgewertet werden.

Dazu wird in einem weiteren Python-Programm die CSV-Datei zunächst in einen Panda Dataframe eingelesen und vor verarbeitet. Diese Daten werden dann mittels Panda, Matplotlib und Seaborn visualisiert, um das Verhalten der verschiedenen Abfragen und vor allem der verschiedenen Abfragearten analysieren zu können.

5 Ergebnis

Das Ziel dieses Diplomarbeitsprojekt ist es, dem Unternehmen ITPRO eine schnelle und zielgerechte Hilfestellung für die Suche in großen Dokumenten-Ordnern zu erschaffen. Dabei gilt es als Grundvoraussetzung, die erstellte Software in die unternehmenseigenen Systeme integrieren zu können.

Das Ergebnis ist ein RAG-System, welches aus einem Frontend, einem Backend und zwei Datenbanken besteht. Die Software nutzt die Authentifizierungsinfrastruktur der ITPRO, um Zugriffskontrollen über SSO abzuwickeln.

Ebenfalls zum Ergebnis gehören die Analysen des Hauptprozesses zum Vergleich der verwendeten Datenbanken aufgrund ihrer Leistung und Performance.

5.1 Analyseergebnisse

Die Analysen wurden mit dem eigens geschriebenen Hilfsprogramm Analyzer durchgeführt. Wie dieser funktioniert und wie die Analysen damit durchgeführt wurden, wird in diesem Kapitel beschrieben: Analyzer

5.1.1 Datenbank Analyseergebnis Elasticsearch

Ein großer Teil der Diplomarbeit war die Optimierung und der Vergleich der Datenbanken Elasticsearch und Milvus. Insbesondere der Unterschied zwischen Elasticsearch als lexikalische Datenbank und der Vektordatenbank Milvus interessierte uns, da es sich um zwei völlig unterschiedliche Abfragemethoden handelt.

Gesamtabfragezeit zwischen Elasticsearch und Milvus im Vergleich:

Um den zeitlichen Unterschied für einen Anwender des Endproduktes zu analysieren, wurde ein Vergleich der Gesamtabfragezeiten der beiden Datenbanken vorgenommen. Dieser Vergleich umfasst die Verarbeitung der Frage in einen Query bzw. in Vektoren zur Abfrage auf die Datenbank, die Suche in der Datenbank und das Generieren einer natürlichen Antwort durch das LLM. Die Ergebnisse sind in Abbildung 62 dargestellt. Es zeigt sich, dass sich die Gesamtdauer,

bis eine Antwort auf die Frage erhalten wird, trotz der Vorteile der Vektordatenbank bei der Suchzeit (siehe hier) zwischen den Datenbanken nur geringfügig unterscheidet.

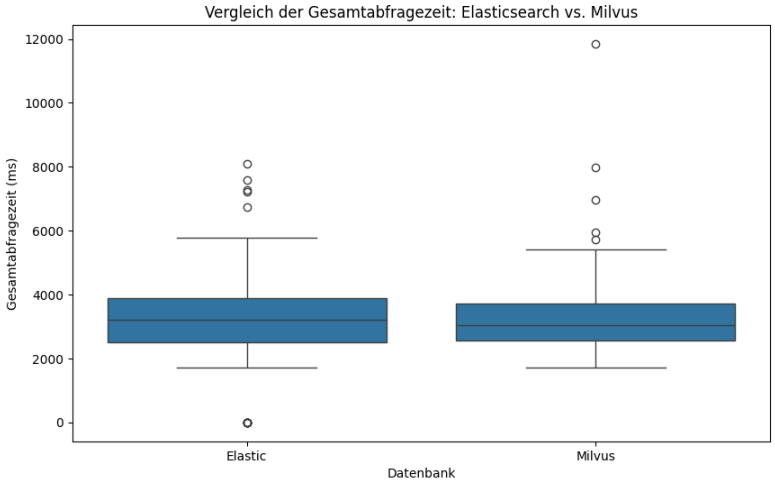


Abbildung 62: Vergleich der Gesamtabfragezeiten

Gesamtabfragezeiten zwischen den Elasticsearch Abfragemethoden im Vergleich:

Wie auch schon im Vergleich zwischen den beiden Datenbanken, ist die durchschnittliche Gesamtabfragedauer auch zwischen den drei getesteten Elasticsearch Abfragearten relativ gleich. Interessant wird es aber, wenn man sich den Bereich des *defaultQueries* ansieht. Beim *defaultQuery* ist deutlich zu erkennen, dass sich der Bereich viel weiter nach unten erstreckt. Dies ist leider nicht damit zu begründen, dass diese Abfrageart besonders effektiv und schnell ist, sondern oft Queries erstellt wurden, die kein Ergebnis aus der Datenbank lieferten und somit keine natürliche Antwort generiert werden musste, was natürlich einige Zeit spart.

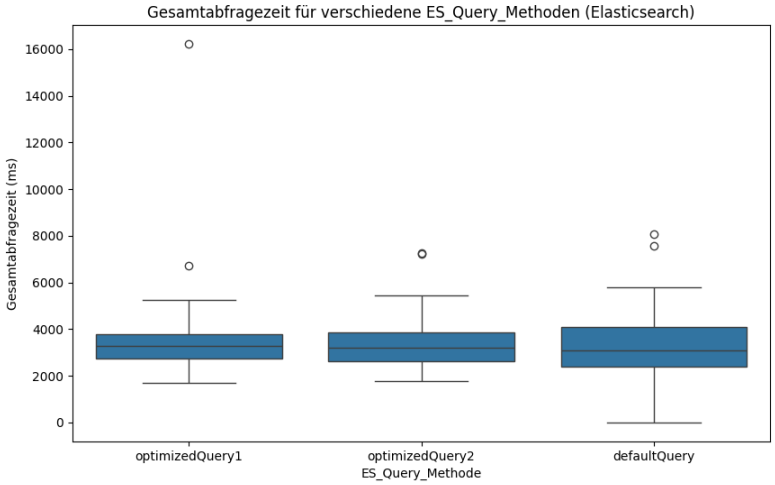


Abbildung 63: Gesamtabfragezeit nach Elasticsearch Abfragemthoden

5.1.2 Datenbank Analyseergebnis Milvus

Suchzeit zwischen Elasticsearch und Milvus im Vergleich:

Der folgende Boxplot zeigt den Vergleich der Suchzeit zwischen Elasticsearch und Milvus.

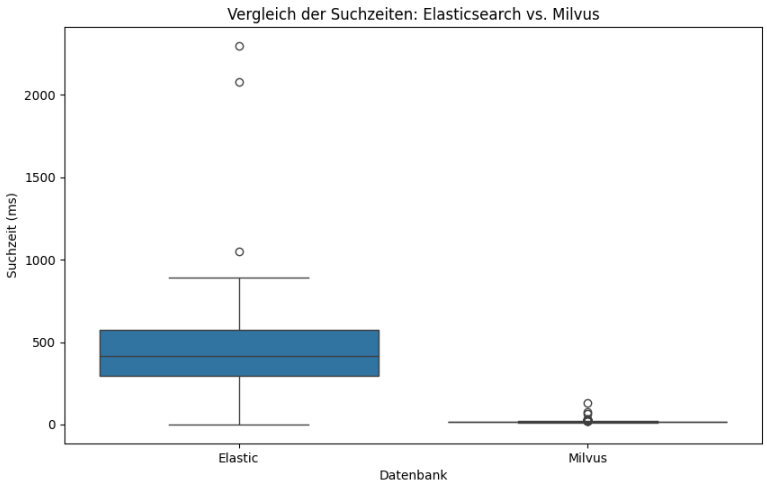


Abbildung 64: Vergleich der Suchzeiten zwischen Elasticsearch und Milvus

Die Analyse zeigt, dass es zwischen den beiden Datenbanken große Unterschiede in der Suchzeit gibt. Bei **Elasticsearch** variieren die Suchzeiten stark, der Median liegt bei etwa 400 bis 500 Millisekunden. Zudem treten mehrere Ausreißer auf, die teilweise über 2000 Millisekunden hinausgehen.

Milvus hingegen überzeugt durch seine konstant schnellen Suchzeiten, was auch deutlich in der Grafik zu erkennen ist. Der Median liegt hier bei rund 50 Millisekunden. Die wenigen Ausreißer bei Milvus sind im Vergleich deutlich geringer und überschreiten nicht 200 Millisekunden.

Korrektheit der Antwort in Milvus:

Das folgende Kreisdiagramm zeigt die Ergebnisse von Milvus ohne den Einsatz von LLM-Contents. 68,3 Prozent der Suchtreffer sind vollständig korrekt. In 28,3 Prozent der Fälle war sowohl das gefundene Dokument als auch der Inhalt der Antwort falsch. Nur 3,3 Prozent der Ergebnisse enthielten zwar das falsche Dokument, aber der Inhalt der Frage war korrekt.

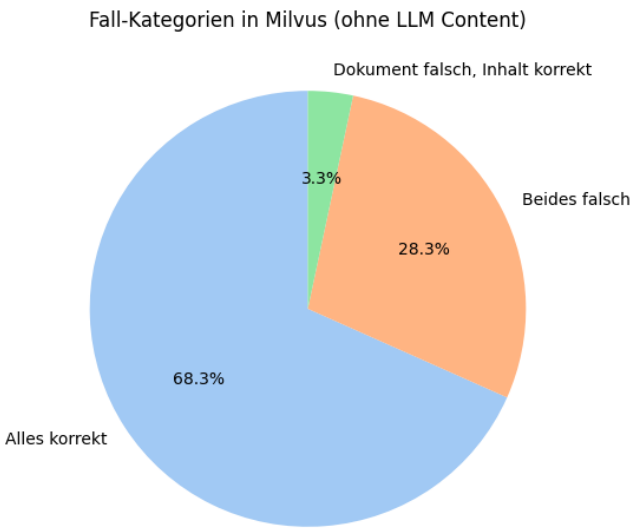


Abbildung 65: Verteilung der Fall-Kategorien in Milvus ohne LLM-Content

Im zweiten Kreisdiagramm sind die Ergebnisse mit LLM-Contents dargestellt. Hier steigt die Genauigkeit deutlich, 72,2 Prozent der Treffer sind komplett korrekt. Die Anzahl der komplett falschen Ergebnisse sinkt auf 23,3 Prozent. Der Anteil der Antworten mit korrektem Inhalt, aber falschem Dokument steigt auf 4,4 Prozent an.

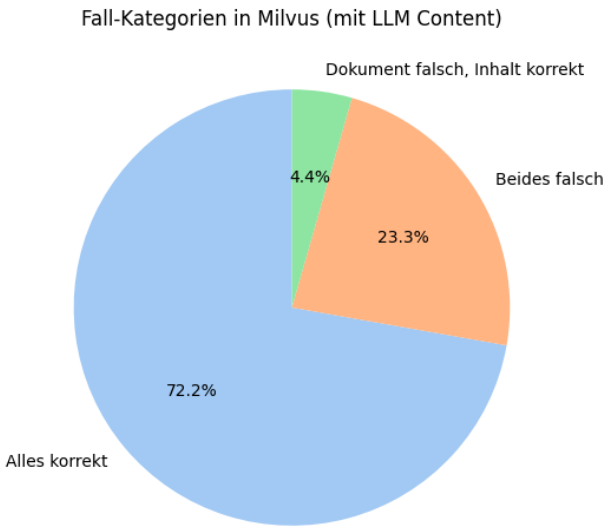


Abbildung 66: Verteilung der Fall-Kategorien in Milvus mit LLM-Content

Durch die Verwendung von LLM-Contents liefert Milvus bessere und verlässlichere Ergebnisse. Es gibt mehr korrekte Antworten und weniger fehlerhafte Ausgaben, was für eine verbesserte semantische Suche spricht.

Korrektheit der Antworten in Elasticsearch

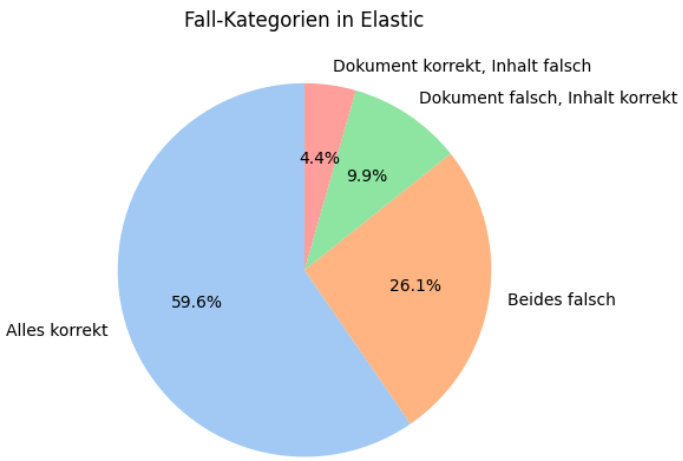


Abbildung 67: Prozentuale Korrektheit in Elasticsearch

Interessant wird es, wenn man sich anschaut, wie sich die prozentuale Korrektheit auf die verschiedenen Abfragemethoden auswirkt, insbesondere im Vergleich mit der Verwendung eines LLM-Contents.

Es fällt sofort auf, dass der anders generierte *defaultQuery* deutlich schlechtere Ergebnisse liefert als die beiden optimierten Queries. Bemerkenswert ist, dass der *defaultQuery* unter Verwendung eines LLM-Contents sogar schlechtere Ergebnisse liefert, was dem Sinn und Zweck von LLM-Contents entgegensteht.

Zu sehen ist auch, dass sich die ohnehin schon bemerkenswert guten Ergebnisse der optimierten Suchanfragen durch das Hinzufügen bestimmter Zusatzinformationen durch die LLM-Contents noch einmal deutlich verbessern lassen. Dies ist darauf zurückzuführen, dass bei den LLM-Contents in den Analysen darauf hingewiesen wurde, dass trotz der in deutscher Sprache gestellten Frage, die Dokumente, in denen gesucht werden soll, auf Englisch sein können oder auch andere Wörter für dieselben Inhalte verwenden worden sein könnten.

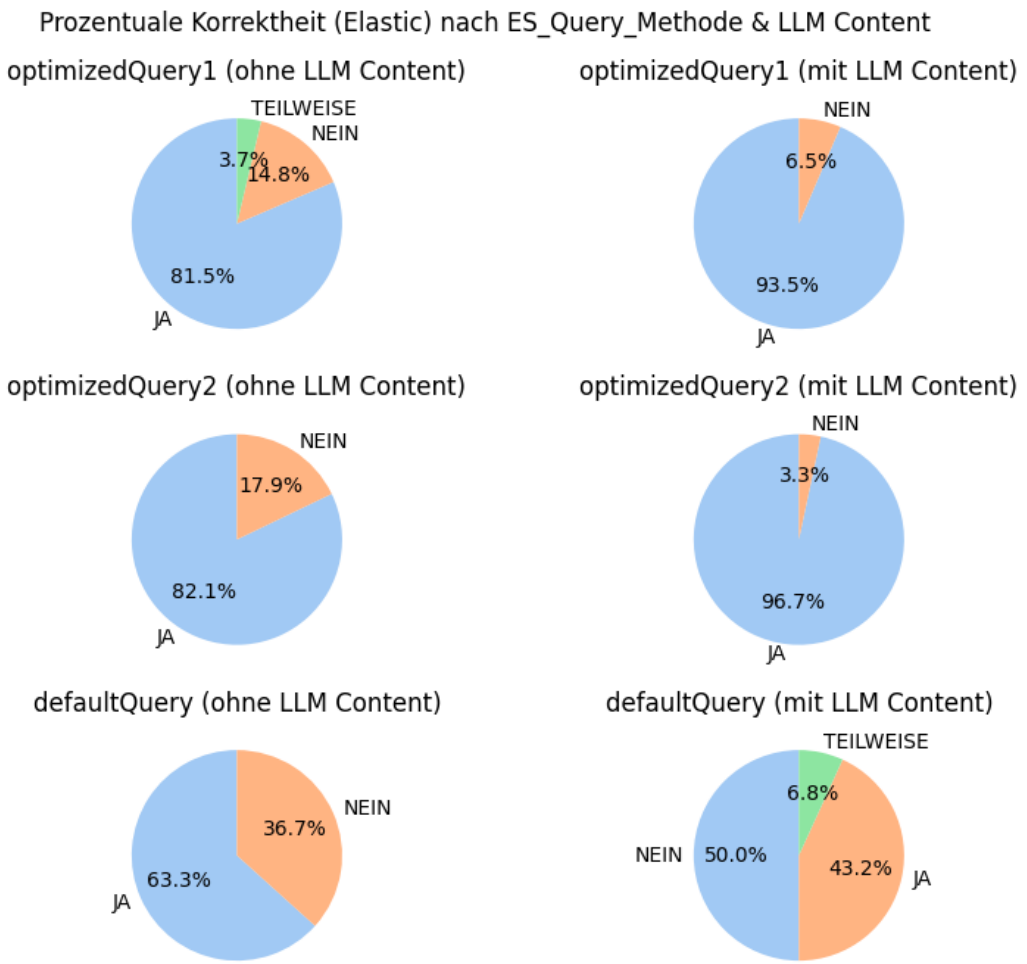


Abbildung 68: Prozentuale Korrektheit nach Abfragemethode und LLM-Content

5.2 Ergebnisse Backend

Das Backend übernimmt den Prozess der Datenbeschaffung sowie den Hauptprozess. Ergebnis ist die Datenaufbereitung und Einarbeitung von Dokumenten in Milvus und Elasticsearch, sowie die Bereitstellung des Frageantwortprozesses.

Mittels einer automatischen Extraktion lädt der **DocumentDissector** Dokumente aus einem ausgewählten Ordner und transformiert die Daten für die Speicherung in ein einheitliches Format. Dieser Service speichert die Dokumente und deren Metadaten effizient in den Datenbanken Elasticsearch und Milvus und überwacht die Ordner auch automatisch auf Änderungen.

Die zweite Service-Anwendung stellt Endpunkte nach außen bereit und führt den **Frageantwortprozess** durch. Entscheidend ist dabei das **LLM** der **OpenAI-API**. Dieses Modell führt zu sehr zufriedenstellenden Antworten, wenn die richtigen Prompts angewendet werden. Im Rahmen des Projektbetriebs wurde auch ein Versuch gestartet, ein LLM von Ollama lokal zu hosten, es gilt aber zu akzeptieren, dass ein Rechner alleine dafür nicht genügend Leistung bietet. Der Frageantwortprozess war in diesem Szenario ineffizient, langsam und führte oft zu qualitativ niedrigen Antworten.

Ein weiteres Ergebnis sind die **Konfigurationsklassen** der **Suchdatenbanken** und die damit verbundenen **Vorlagen** und **Prompts**, die während des Projektbetriebs erweitert und verbessert worden sind. Einzelne Recherchen und Tests haben zur Verbesserung der Prompt-Techniken geführt. Das System verfügt zum Projektabschluss über einige vorgefertigte Texte, die für verschiedene Zwecke individuell gut geeignet sind. Die zweite Art, das Einfügen von Schlüsselwörtern in eine Vorlage, ist ebenfalls verbessert worden.

Wird der Prozess mit der Bauanleitung gewählt, gibt es den Vorteil, dass Elasticsearch einen eigenen Query erstellen könnte. Dieser kann auch vom Suchtyp bei jeder Abfrage unterschiedlich sein und ist somit individueller. Beim Prozess mit der **Vorlage** sind in unserem Projektbetrieb **bessere Ergebnisse** herausgekommen. Das manuelle Einfügen von Schlüsselwörtern führt dazu, dass der Query verlässlich in der gleichen Form bleibt. Somit können keine Fehler von OpenAI ins System eingeschleust werden. Weitere Ergebnisse und Analysen dazu sind im Abschnitt 5.1 nachzulesen.

Da im RAG-System eine große Anzahl von Komponenten voneinander abhängig sind und die korrekte Konfiguration sehr wichtig ist, ist ein **Dokument** für die **Einrichtung** des Systems erstellt worden. Dieses Dokument ist dem Unternehmen bereitgestellt worden, um das System auch nach Abschluss der Diplomarbeit weiterverwenden zu können.

5.3 Benutzeroberfläche

Die Benutzeroberfläche ist eine Angular-Anwendung zur Verwaltung der Queries eines Benutzers.

Der Zugriff auf die Webanwendung ist nur nach einer erfolgreichen Anmeldung über das SSO von ITPRO möglich. Nach der Anmeldung wird durch Silent-Refresh der Access-Token, der für den Zugriff auf die Anwendung erforderlich ist, automatisch erneuert, bevor er abläuft.

Nach erfolgreicher Anmeldung hat der Benutzer Zugriff auf die Hauptseite. Auf der Hauptseite werden die gespeicherten Chats des angemeldeten Benutzers angezeigt, die jederzeit gelöscht oder umbenannt werden können. Vom ausgewählten Chat werden die gespeicherten Queries des Chats auf der Seite angezeigt.

Der Benutzer kann auch neue Fragen stellen und hat dazu die Möglichkeit, die Antwort gezielt zu beeinflussen, indem er einen der vorhandenen LLM-Contents und die Suchmethode auswählt.

6 Resümee

Die Suche nach Informationen in großen Sammlungen von Dokumenten ist eine herausfordernde Aufgabe, welche mithilfe von künstlicher Intelligenz bewältigt werden kann. Die Retrieval-Augmented Generation bietet sich dafür sehr gut an, da damit der Fokus der Beantwortung auf einem Abschnitt eines Dokuments liegt.

Essenziell für diese Methodik ist der Suchvorgang. Die Qualität des Systems hängt maßgeblich davon ab, ob der richtige Teil eines Dokuments gefunden wird. Ohne Zusatzeinstellungen sind anfangs bei beiden Datenbanken mit mindestens 75 Prozent die korrekten Antworten geliefert worden. Dieses Ergebnis hat sich durch die Optimierung und Verwendung eines zusätzlichen Kontexts (LLM_Content) noch einmal maßgeblich verbessert.

Elasticsearch bietet im System die bessere Qualität in der Beantwortung. Gestellte Fragen mit einem LLM_Content sind nach der Optimierung zu rund 95 Prozent korrekt beantwortet worden. Die Datenbank Milvus hängt mit der Korrektheit zurück. Ein deutlicher Unterschied ist bei der Suchzeit zu erkennen. Während Elasticsearch einen Median von rund 400 Millisekunden in der Suche hat, hat Milvus einen Median von 50 Millisekunden. Mit der zweiten Datenbank sind also weitaus mehr Suchvorgänge pro Sekunde möglich.

Gemeinsam ist ein RAG-System entwickelt worden, welches in die Unternehmensumgebung eingebettet werden kann. Mitarbeiter und Kunden haben die Möglichkeit, über das Frontend die Funktionalität zu nutzen, dabei ist kein Hintergrundwissen notwendig. Das System kann auf den Servern des Unternehmens gehostet werden. Durch die implementierten Sicherheitseinstellungen sind die Daten geschützt.

Das Projektteam hat sich während der Entwicklung spezialisiertes Wissen in verschiedenen Teilbereichen der Informatik aneignen können. Milvus und Elasticsearch sind moderne Datenbanken und werden von namhaften Firmen im Produktivbereich eingesetzt. Ebenfalls gewinnt die Verwendung von KI in eigens entwickelten Systemen immer mehr an Bedeutung.

Die Retrieval-Augmented Generation wird für Unternehmen aller Branchen wichtig, da die Menge gespeicherter Dokumente in den kommenden Jahren weiterhin exponentiell wachsen wird. Firmen können damit die Herausforderung meistern, Wissen effizient zu organisieren und nutzbar zu machen.

7 Aufgabenverteilung

Im folgenden Punkt ist festgehalten, wer welche Kapitel der Diplomarbeit verfasst hat. Teilweise wurden Kapitel von mehreren Autoren gemeinsam verfasst, da der jeweilige Bereich in die Zuständigkeit mehrerer oder aller Teammitglieder fällt.

7.1 Sebastian Horner

Inhaltsverzeichnis Sebastian

7.2 Moritz Kern

Inhaltsverzeichnis Moritz

7.3 Martin Pilgerstorfer

Inhaltsverzeichnis Martin

7.4 Tobias Wahl

Inhaltsverzeichnis Tobias

Glossar & Abkürzungsverzeichnis

API application programming interface

bzw. beziehungsweise

CRUD create - read - update - delete

DBMS Datenbankmanagementsystem

JWT JSON Web Token

LLM Large Language Model

MVC Model-View-Controller

OCR optical character recognition

ORM Object relational mapping

RAG Retrieval Augmented Generation

REST Representational State Transfer

SPA Single-Page Applications

URL Uniform Resource Locator

usw. und so weiter

uvm. und vieles mehr

v. l. n. r. von links nach rechts

WSL Windows Subsystem for Linux

Literaturverzeichnis

[1] L. Wuttke, „Machine Learning vs. Deep Learning: Wo ist der Unterschied?" 2025, letzter Zugriff am 05.02.2025. Online verfügbar: https://datasolut.com/machine-learning-vs-deep-learning/

[2] I. Ltd., „What are LLM Hallucinations?" 2025, letzter Zugriff am 05.02.2025. Online verfügbar: https://www.iguazio.com/glossary/llm-hallucination/

[3] P. Lewis, E. Perez *et al.*, „Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," 2025, letzter Zugriff am 05.02.2025. Online verfügbar: https://arxiv.org/abs/2005.11401

[4] A. S. Gillis, „What is retrieval-augmented generation (RAG) in AI?" 2025, letzter Zugriff am 05.02.2025. Online verfügbar: https://www.techtarget.com/searchenterpriseai/definition/retrieval-augmented-generation

[5] D. Brimley, „Invertierte Indizes für Textsuche/lexikalische Suche," 2023, letzter Zugriff am 20.02.2025. Online verfügbar: https://www.elastic.co/de/blog/what-is-an-elasticsearch-index#invertierte-indizes-f%C3%BCr-textsuche/lexikalische-suche

[6] Duden, „lexikalisch," 2025, letzter Zugriff am 04.02.2025. Online verfügbar: https://www.duden.de/rechtschreibung/lexikalisch

[7] T. Cloer, „Semantisch," 2024, letzter Zugriff am 19.02.2025. Online verfügbar: https://de.wikipedia.org/wiki/Semantische_Suche

[8] Wikipedia, „Python (programming language)," 2025, letzter Zugriff am 20.02.2025. Online verfügbar: https://en.wikipedia.org/wiki/Python_(programming_language)

[9] ——, „Python (Programmiersprache)," 2025, letzter Zugriff am 20.02.2025. Online verfügbar: https://de.wikipedia.org/wiki/Python_(Programmiersprache)

[10] A. Community, „What is Angular?" 2025, letzter Zugriff am 06.03.2025. Online verfügbar: https://angular.dev/overview

[11] Wikipedia, „Docker (software)," 2025, letzter Zugriff am 20.02.2025. Online verfügbar: https://en.wikipedia.org/wiki/Docker_(software)

[12] ——, „Docker (Software)," 2025, letzter Zugriff am 20.02.2025. Online verfügbar: https://de.wikipedia.org/wiki/Docker_(Software)

[13] ——, „Windows Subsystem for Linux," 2025, letzter Zugriff am 20.02.2025. Online verfügbar: https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux

[14] I. Redaktion, „WSL2: Vorstellung des Windows Subsystems für Linux 2," 2022, letzter Zugriff am 20.02.2025. Online verfügbar: https://www.ionos.de/digitalguide/server/knowhow/wsl2-vorgestellt/

[15] Wikipedia, „Windows-Subsystem für Linux," 2025, letzter Zugriff am 20.02.2025. Online verfügbar: https://de.wikipedia.org/wiki/Windows-Subsystem_f%C3%BCr_Linux

[16] M. 2025, „Übersicht über ASP.NET Core," 2025, letzter Zugriff am 02.04.2025. Online verfügbar: https://learn.microsoft.com/de-de/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-9.0

[17] Elastic, „REST APIs," 2025, letzter Zugriff am 30.03.2025. Online verfügbar: https://www.elastic.co/guide/en/elasticsearch/reference/current/rest-apis.html

[18] M. Ingebrigtsen, „Indexing for Beginners, Part 2," 2013, letzter Zugriff am 30.03.2025. Online verfügbar: https://www.elastic.co/blog/found-indexing-for-beginners-part2

[19] ——, „Indexing for Beginners, Part 3," 2013, letzter Zugriff am 30.03.2025. Online verfügbar: https://www.elastic.co/blog/found-indexing-for-beginners-part3

[20] Milvus, „Milvus-Doc," 2025, letzter Zugriff am 01.03.2025. Online verfügbar: https://milvus.io/docs/de/overview.md

[21] ——, „milvus-architektur," 2025, letzter Zugriff am 01.03.2025. Online verfügbar: https://milvus.io/docs/de/v2.4.x/architecture_overview.md

[22] S. Patel, A. Svyryd *et al.*, „Entity Framework Core," 2025, letzter Zugriff am 06.02.2025. Online verfügbar: https://learn.microsoft.com/en-us/ef/core/

[23] Wikipedia, „Visual Studio Code," 2025, letzter Zugriff am 20.02.2025. Online verfügbar: https://en.wikipedia.org/wiki/Visual_Studio_Code

[24] ——, „Visual Studio Code," 2025, letzter Zugriff am 20.02.2025. Online verfügbar: https://de.wikipedia.org/wiki/Visual_Studio_Code

[25] JetBrains, „Meet WebStorm," 2025, letzter Zugriff am 06.03.2025. Online verfügbar: https://www.jetbrains.com/help/webstorm/meet-webstorm.html

[26] Wikipedia, „Postman (software)," 2025, letzter Zugriff am 21.02.2025. Online verfügbar: https://en.wikipedia.org/wiki/Postman_(software)

[27] P. Harvey, „ExifTool by Phil Harvey | Read, Write and Edit Meta Information!" 2025, letzter Zugriff am 21.02.2025. Online verfügbar: https://exiftool.org/

[28] GeeksforGeeks, „What is Angular Material?" 2024, letzter Zugriff am 16.02.2025. Online verfügbar: https://www.geeksforgeeks.org/what-is-angular-material/

[29] NGxS, „NGXS," 2024, letzter Zugriff am 25.03.2025. Online verfügbar: https://www.ngxs.io

[30] ——, „NGXS Overview," 2023, letzter Zugriff am 25.03.2025. Online verfügbar: https://www.ngxs.io/~gitbook/image?url=https%3A%2F%2F2789922418-files.gitbook.io%2F%7E%2Ffiles%2Fv0%2Fb%2Fgitbook-x-prod.appspot.com%2Fo%2Fspaces%252F-L9CoGJCq3UCfKJ7RCUg-347405460%252Fuploads%252Fgit-blob-7371002ded66c4455ca986a4c8e7c1f6849ffef9%252Fdiagram.png%3Falt%3Dmedia&width=768&dpr=1&quality=100&sign=45d8b3d9&sv=2

[31] G. Brockman, M. Murati *et al.*, „OpenAI API," 2025, letzter Zugriff am 05.02.2025. Online verfügbar: https://openai.com/index/openai-api/

[32] csharp411, „ml.net-logo," 2025, letzter Zugriff am 01.03.2025. Online verfügbar: https://www.csharp411.com/wp-content/uploads/2022/08/ML.NET-Logo.wine_.png

[33] ACS, „Was ist SSO und wann wird es benötigt?" 2023, letzter Zugriff am 16.02.2025. Online verfügbar: https://www.acs.it/de/blog/cybersicherheit/sso-single-sign-on

[34] Sebastian Peyrott, „What Is Single Sign-On Authentication (SSO) And How Does It Work?" 2023, letzter Zugriff am 16.02.2025. Online verfügbar: https://auth0.com/blog/what-is-and-how-does-single-sign-on-work/

[35] Ryan Greene, Ted Sanders, Lilian Weng, Arvind Neelakantan, „Text-embedding-model," 2022, letzter Zugriff am 15.02.2025. Online verfügbar: https://openai.com/index/new-and-improved-embedding-model/

[36] Milvus, Zilliz, „Vektorsuche," 2025, letzter Zugriff am 15.02.2025. Online verfügbar: https://milvus.io/docs/de/vector_visualization.md

[37] R. Merrit, „What Is Retrieval-Augmented Generation, aka RAG?" 2025, letzter Zugriff am 05.02.2025. Online verfügbar: https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/

[38] T. Dykstra, L. Latham *et al.*, „ASP.NET Core Middleware," 2025, letzter Zugriff am 05.02.2025. Online verfügbar: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-9.0

[39] miro, „miro," 2025, letzter Zugriff am 01.03.2025. Online verfügbar: https://miro.com/app/dashboard/

[40] etcd, „Funktion von etcd," 2025, letzter Zugriff am 18.02.2025. Online verfügbar: https://etcd.io/

[41] MinIO, „MinIO-Milvus," 2025, letzter Zugriff am 18.02.2025. Online verfügbar: https://min.io/docs/minio/kubernetes/upstream/index.html

[42] M. Steyer, „Single Sign-on mit OAuth 2 und OpenId Connect," 2020, letzer Zugriff am 03.02.2025. Online verfügbar: https://angular.de/artikel/oauth-odic-plugin/

[43] T. Schaz, „OAuth 2.0: Der Authorization Code Flow im Detail," 2022, letzer Zugriff am 03.02.2025. Online verfügbar: https://blog.doubleslash.de/developer-blog/oauth-2-0-der-authorization-code-flow-im-detail

[44] https://www.npmjs.com/package/angular-oauth2-oidc#Thanks-to-all Contributors, „angular-oauth2-oidc," 2024, letzter Zugriff am 04.02.2025. Online verfügbar: https://www.npmjs.com/package/angular-oauth2-oidc

[45] Angular, „Common Routing Tasks," 2025, letzter Zugriff am 16.02.2025. Online verfügbar: https://angular.dev/guide/routing/common-router-tasks

[46] S. Burgemeister, „Was ist Prompting?" 2025, letzter Zugriff am 09.03.2025. Online verfügbar: https://www.iu-akademie.de/blog/was-ist-prompting/

[47] S. M. Michael Günther, „Query-Expansion mit LLMs: Bessere Suchergebnisse durch ausführlichere Anfragen," 2025, letzter Zugriff am 09.03.2025. Online verfügbar: https://jina.ai/de/news/query-expansion-with-llms-searching-better-by-saying-more/

Abbildungsverzeichnis

Anhang

A Ursprüngliche Projektidee

1. OPENAI FÜR DOKUMENTENDURCHSUCHUNG

Es soll möglich sein, mit natürlicher Sprache Dokumente zu durchsuchen und im Idealfall eine Text-Antwort auf eine Frage zu erhalten.
Hierzu sollen verschiedene (zu definierende, Word, PDF, HTML, …) Dateien automatisch indiziert werden (z. B. ElasticSearch) und als Datenquelle dienen. Die indizierten Daten sollen dann mittels einer Web-Schnittstelle (REST, oder ähnlich kompatibles) durchsucht werden können.
Als Eingabemaske gibt es nur ein Eingabefeld, wo hier eine Frage eingegeben werden kann. Durch die Verwendung von OpenAI soll eine gute Suchanfrage für das Such-Service erstellt werden (z. B. über Function calling and other API updates (openai.com)). Dieses soll dann mittels .NET (C#)/Java mit dem Ergebnis von OpenAI aufgerufen werden.

- In Ausbaustufe 1 sollen die Ergebnisse dann aufgelistet werden (inkl. Vorschau wo die Einträge gefunden wurden und in welchem Dokument gefunden wurde).
- In Ausbaustufe 2 soll über OpenAI eine konkrete Antwort auf die gestellte Frage erstellt werden. Hierfür können die Suchergebnisse auch an OpenAI geschickt werden, damit daraus die Antwort erstellt werden kann.

1.1. TECHNOLOGIEN

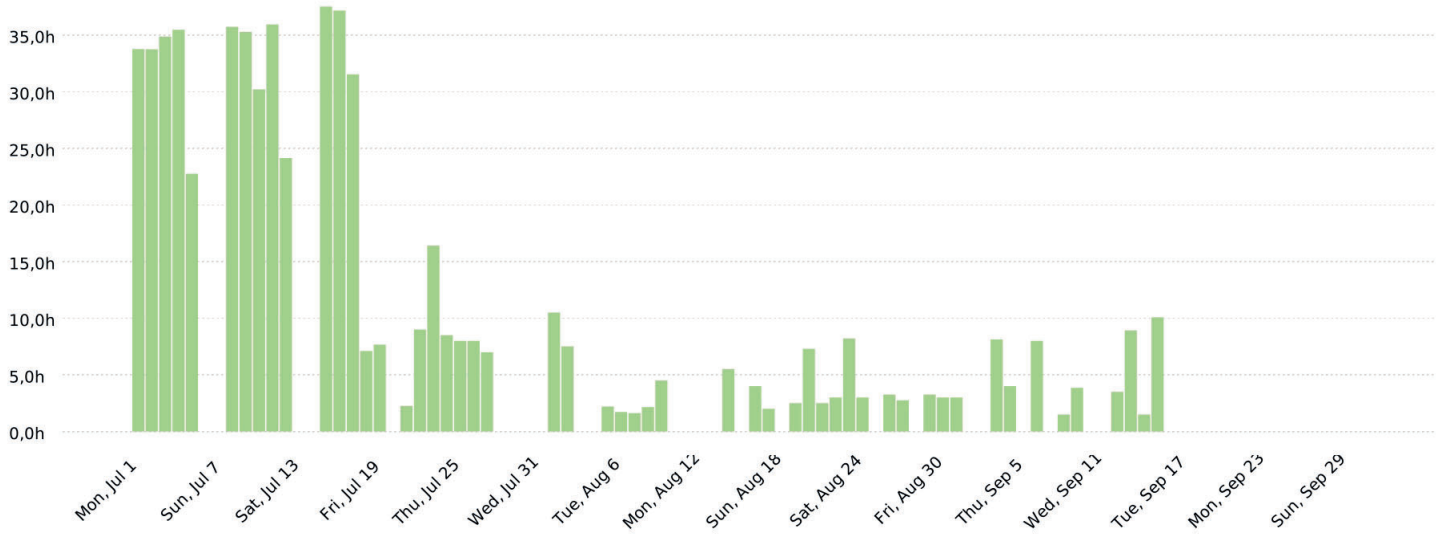
OpenAI, REST, .NET/Java (nach Belieben), ev. Angular (für Eingabemaske)

B Clockify Zeitaufzeichnung

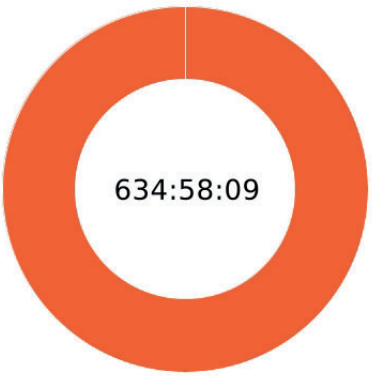
Summary report

01/07/2024 - 30/09/2024

Total: 634:58:09



Project



DA	634:58:09	100,00%

C Besprechungsprotokolle des gesamten Projekts

Diese Besprechungsprotokolle wurden zum Teil schon während des Projektes der 4. Klasse, auf dem diese Diplomarbeit aufbaut, erstellt, da dort zum Teil schon die Vorgehensweise der Diplomarbeit besprochen wurde.

Sitzungsprotokoll zum KI-Projekt

Sebastian Horner

16. Mai 2024

Anwesend:

Name	Rolle
Sebastian Horner	Projektkoordinator & Database-Engineer
Moritz Kern	Schriftführer & Frontend-Entwickler
Martin Pilgerstorfer	Pressesprecher & Backend-Entwickler
Tobias Wahl	Qualitätsmanager & Backend-Entwickler
Delia Dorn	Projektbetreuerin ITPro
David Aberl	Projektbetreuer (techn.) ITPro

Entschuldigt:

/

Protokollführer

Sebastian Horner

Inhaltsverzeichnis

1 Fragen

2 Hauptpunkte

2.1 Backend

Martin hat Davids Vorschläge (Ausbesserungen) umgesetzt. Letzte Methoden sollten auf Async umgestellt werden. OpenAI configs sollten in den IOptions sein. appsettings.json

2.2 Frontend

CSS wiederhergestellt (repariert). Optisch aufhübschen! Schönere Schriftart!

2.3 Database

Dateien werden nur zu Beginn indiziert. Wenn Dateien umgeändert werden, reindex funktionsfähig?

2.4 API

OpenAI sollte Äntwortünd "Frage"nicht zurückgeben bzw. sollte das nicht sondern im Backend. Evntl im Kontext an OpenAI übergeben, ansonsten regex. Im Kontext angeben, dass in der selben Sprache geantwortet wird wie gefragt.

2.5 Allfälliges

Arbeitsbeginn, Montag 01. Juli 2024 8:00 Uhr, brauchen nichts mitnehmen, evtl Jause, es gibt eine Kantine, alle anderen Tage Gleitzeit mit Kernarbeitszeit von 9:00 Uhr bis 15:00 Uhr.

Sitzungsende

16. Mai 2024, 11:30 Uhr

Protokollführer:	Sebastian Horner	
Unterschrift:		

Sitzungsprotokoll zum KI-Projekt

Sebastian Horner

23. April 2024

Anwesend:

Name	Rolle
Sebastian Horner	Projektkoordinator & Database-Engineer
Moritz Kern	Schriftführer & Frontend-Entwickler
Martin Pilgerstorfer	Pressesprecher & Backend-Entwickler
Tobias Wahl	Qualitätsmanager & Backend-Entwickler
Delia Dorn	Projektbetreuerin ITPro
David Aberl	Projektbetreuer (techn.) ITPro

Entschuldigt:

/

Protokollführer

Sebastian Horner

Inhaltsverzeichnis

1 Fragen

2 Hauptpunkte

2.1 Indizierung mit FSCrawler in Elasticsearch

Error-Code vom letzten mal wurde behoben, außerdem wurde das docker-compose so umgeändert, dass relative Pfade verwendet werden und somit von jedem verwendet werden kann.

Indizieren von Webanwendungen, indizierung von Word,PPT,Excel-Dokumenten etc..

2.2 Backend

David hat den Code überarbeitet. Bei der AnsweringUnit soll mit Interfaces gearbeitet werden. IAnsweringUnit und mittels Dependency-Injection wird die Implementierung hinzugefügt. Verschiedene Lifetimes (Singleton, Scoped, Stateless). ASP.NET arbeitet nach außen mit JSON-Objekten durch den Controller. Async Methoden Naming-Conventions anwenden. Generell sollte auf Naming Conventions geachtet werden. Einstellungen sollten nicht Hardcoded sein (Appsettings, IOption, configure-methoden)

2.3 Frontend

Hat jetzt Eingabefeld für verschiedene LLM-Methoden. CSS ist gerade kaputt (alles kaputt). Bei der Antwort wird oft die Frage nocheinmal wiederholt.

2.4 Allfälliges

Es wurde sich über den aktuellen Stand ausgetauscht und das weitere vorgehen besprochen. Außerdem wurde der Diplomarbeitsantrag für das ABA-Portal durchgeschaut.

3 Nächste Sitzung

23. April 2024, 08:00 Uhr

Sitzungsende

16. Mai 2024, 11:00 Uhr

Protokollführer:	Sebastian Horner	
Unterschrift:		

Sitzungsprotokoll zum KI-Projekt

Sebastian Horner

21. März 2024

Anwesend:

Name	**Rolle**
Sebastian Horner	Projektkoordinator & Database-Engineer
Moritz Kern	Schriftführer & Frontend-Entwickler
Martin Pilgerstorfer	Pressesprecher & Backend-Entwickler
Tobias Wahl	Qualitätsmanager & Backend-Entwickler
Delia Dorn	Projektbetreuerin ITPro
David Aberl	Projektbetreuer (techn.) ITPro

Entschuldigt:

/

Protokollführer

Sebastian Horner

Inhaltsverzeichnis

1 Fragen

1.1 Sollen noch Verbesserungen bei der Indizierung vorgenommen werden?

Zuerst Fehler beheben, in wie wwit kann man die Indizierung beeinflussen und was hat das für eine auswirkung.

2 Hauptpunkte

2.1 Indizierung mit FSCrawler in Elasticsearch

Die Indizierung mittels FSC, welche nun funktioniert, wurde vorgestellt. Der noch vorhandene Error-Code wird von Sebastian noch bearbeitet.

2.2 Backend

Vorstellung des aktuellen Standes im Backend (ein Request welcher im Frontend eingegeben wird kann bereits verarbeitet werde, wird aber noch nicht im Frontend angezeigt). Eindruck von David ist das inhaltlich serh viel weitergegangen ist, codemäßig gehört noch einiges gemacht, bei einem akzeptablen Zwischenstand können wir mit David gerne nochmal über den Code rübergehen (Pull Request bzw. Meeting oder E-Mail).

2.3 Frontend

Evtl. sollen Konfigurationen im Frontend zur Abfrage möglich sein. Z.B. einen Kontext für ChatGPT.

2.4 git

Das Mergen/pullen von c# funktioniert nicht. Das .gitignore sieht nicht so schlecht aus, der .vs Ordner sollte aber noch vom git entfernt werden (bereits entfernt) und zur .gitignore hinzugefügt werden. Auch für das Frontend sollte ein passendes .gitignore verwendet werden. (auch das gitignore für das Frontend wurde bereits überarbeitet).

2.5 Allfälliges

Es wurde sich über den aktuellen Stand ausgetauscht und das weitere vorgehen besprochen. Für die ausarbeitung des Diplomarbeitsantrags...

3 Nächste Sitzung

23. April 2024, 08:00 Uhr

Sitzungsende

21. März 2024, 12:00 Uhr

Protokollführer:	Sebastian Horner	
Unterschrift:		

Sitzungsprotokoll zum KI-Projekt

Sebastian Horner

25. Jänner 2024

Anwesend:

Name	Rolle
Sebastian Horner	Projektkoordinator & Database-Engineer
Moritz Kern	Schriftführer & Frontend-Entwickler
Martin Pilgerstorfer	Pressesprecher & Backend-Entwickler
Tobias Wahl	Qualitätsmanager & Backend-Entwickler
Delia Dorn	Projektbetreuerin ITPro
David Aberl	Projektbetreuer (techn.) ITPro

Entschuldigt:

/

Protokollführer

Sebastian Horner

Inhaltsverzeichnis

1 Fragen

1.1 Wie soll ein Index ca. aussehen der die Dokumente speichert?

Volltextindex, Metainformationen sind weniger relevant. Unter Volltextindizierung recherchieren wie dies am Besten umgesetzt wird.

2 Hauptpunkte

2.1 Backend

ASP.Net core sinnvolle Variante, dabei ist kein parsing notwendig. Request werden vom Frontend initialisiert, nach reqeust wird response gleich zurückgesendet. Neue ASP.Net core web-api projekt erstellen (darin befindet sich program.cs und controller weather-forecast example).

2.2 Allfälliges

Es wurde sich über den aktuellen Stand ausgetauscht und das weitere vorgehen besprochen.

3 Nächste Sitzung

25. Jänner 2024, 10:00 Uhr

Sitzungsende

14. November 2023, 10:30 Uhr

Protokollführer:	Sebastian Horner	
Unterschrift:		

Sitzungsprotokoll zum KI-Projekt

Sebastian Horner

14. November 2023

Anwesend:

Name	**Rolle**
Sebastian Horner	Projektkoordinator & Database-Engineer
Moritz Kern	Schriftführer & Frontend-Entwickler
Martin Pilgerstorfer	Pressesprecher & Backend-Entwickler
Tobias Wahl	Qualitätsmanager & Backend-Entwickler
Delia Dorn	Projektbetreuerin ITPro
David Aberl	Projektbetreuer (techn.) ITPro

Entschuldigt:

/

Protokollführer

Sebastian Horner

Inhaltsverzeichnis

1 Fragen

1.1 Wie soll ein Index ca. aussehen der die Dokumente speichert?

Volltextindex, Metainformationen sind weniger relevant. Unter Volltextindizierung recherchieren wie dies am Besten umgesetzt wird.

1.2 Wie sollen wir die DevOps verwenden, Repo aktivieren, Meilensteine?

Code ins Repo laden. Readme ausarbeiten. Meilensteine können wir für uns gerne definieren für ITPro weniger relevant.

1.3 Wie sieht's mit dem ApiKey aus?

David informiert sich. (Zum Zeitpunkt der Protokollausarbeitung haben wir den Key bereits von David erhalten und wird auf das Git-Repo geladen werden (Stand: 14. Dezemeber 2023, 11:00 Uhr))

1.4 Sollen die letzten Abfragen gespeichert werden bzw. soll es einen Benutzerlogin geben (als Erweiterung, für die Diplomarbeit)?

Kann bei der Diplomarbeit bzw. im Praktikum eingebaut werden. Es macht Sinn Abfragen und Antworten zu speichern dazu wird natürlich auch eine Authentifikation benötigt werden.

2 Hauptpunkte

2.1 Backend

ASP.Net core sinnvolle Variante, dabei ist kein parsing notwendig. Request werden vom Frontend initialisiert, nach reqeust wird response gleich zurückgesendet. Neue ASP.Net core web-api projekt erstellen (darin befindet sich program.cs und controller weather-forecast example).

2.2 Allfälliges

Es wurde sich über den aktuellen Stand ausgetauscht und das weitere vorgehen besprochen.

3 Nächste Sitzung

25. Jänner 2024, 10:00 Uhr

Sitzungsende

14. November 2023, 10:30 Uhr

Protokollführer:	Sebastian Horner	
Unterschrift:		

Sitzungsprotokoll vom 09. November 2023 zum KI-Projekt

Sebastian Horner

09. November 2023

Anwesend:

Name	Rolle
Sebastian Horner	Projektkoordinator & Database-Engineer
Moritz Kern	Schriftführer & Frontend-Entwickler
Martin Pilgerstorfer	Pressesprecher & Backend-Entwickler
Tobias Wahl	Qualitätsmanager & Backend-Entwickler
Delia Dorn	Projektbetreuerin ITPro
David Aberl	Projektbetreuer (techn.) ITPro

Entschuldigt:

/

Protokollführer

Sebastian Horner, Moritz Kern

Inhaltsverzeichnis

1 Fragen

1.1 Wie differenziert sich die Diplomarbeit zum Projekt?

1.2 Welche Dokumente sollen zum Testen genutzt werden?

2 Hauptpunkte

2.1 Eigenheiten der Diplomarbeit

Bei der Diplomarbeit soll neben der Suche mit Elasticsearch mindestens eine weitere Vektordatenbank implementiert werden. Dabei können dann Verschiedenheiten und Eigenschaften analysiert werden, was weiterführend auch den wissenschaftlichen Teil abdecken würde. Das grundsätzliche Ziel wäre dann herauszufinden, welche Datenbank am Besten geeignet ist.

2.2 Testdaten

Nach Nachfrage wurde der Wunsch geäußert vorerst mit willkürlichen Daten von uns zu testen. Es wird aber auch firmenintern nachgefragt, welche Daten wir zum Testen verwenden können.

2.3 Allfälliges

Wenn der Aufwand zu klein (angen. Diplomarbeit) können verschiedene Datenquellen in die Suche eingefügt werden.

3 Nächste Sitzung

Als erstes Sprint-Meeting wurde eine nächste Sitzung ausgemacht. Nächste Sitzung: 14. Dezember 2023, 10:00 Uhr

Sitzungsende

09. November 2023, 10:30 Uhr

Protokollführer:	Sebastian Horner	Moritz Kern
Unterschrift:		

Sitzungsprotokoll vom 05. Oktober 2023 zum KI-Projekt

Sebastian Horner

05. Oktober 2023

Anwesend:

Name	Rolle
Sebastian Horner	Projektkoordinator & Database-Engineer
Moritz Kern	Schriftführer & Frontend-Entwickler
Martin Pilgerstorfer	Pressesprecher & Backend-Entwickler
Tobias Wahl	Qualitätsmanager & Backend-Entwickler
Delia Dorn	Projektbetreuerin ITPro
David Aberl	Projektbetreuer (techn.) ITPro

Entschuldigt:

/

Protokollführer

Sebastian Horner, Moritz Kern

Inhaltsverzeichnis

1 Fragen

-

2 Hauptpunkte

2.1 Klären der Fragen zum Projekt (Technologie/Umsetzung ...)

Es sollen Suchabfragen aus natürlichem Text mithilfe von OpenAI in eine Abfrage f¨ur Elasticsearch generiert werden. In der ersten Ausbaustufe sollte dann das Ergebnis dieser Abfrage bzw. der Bereich, in dem sich das Ergebnis befindet als Antwort gegeben werden. In Ausbaustufe zwei sollte dieses, noch nicht in einer eindeutigen Antwort formulierte Ergebnis in einen kurzen Satz/kurze Beschreibung formuliert werden. Dies geschieht wiederum wieder mithilfe von OpenAI, welches das Ergebnis der vorherigen Suchabfrage verarbeitet. Der Webclient, in dem die Fragen gestellt werden sollte, ähnlich wie Startseiten bekannter Suchmaschinen aussehen. Im Frontend sollte unbedingt Angular verwendet werden, wobei die Technologie im Backend frei zu wählen ist. Als Datenbank bzw. zum Indexieren der Dokumente sollte Elasticsearch verwendet werden. Dokumente müssen zu Beginn nicht über die Benutzerschnittstelle hinzugefügt werden. Am Besten wäre dafür ein Verzeichnis welches in Elasticsearch indexiert wird und bei

2.2 Allfälliges

-

3 Nächste Sitzung

Um das weitere Vorgehen zu besprechen, wurde ein weiterer Besprechungstermin vereinbart. Nächste Sitzung: 09. November 2023, 10:00 Uhr

Sitzungsende

05. Oktober 2023, 11:30 Uhr

Protokollführer:	Sebastian Horner	Moritz Kern
Unterschrift:		