



TECHNISCHE HOCHSCHULE MITTELHESSEN

**THM**

**CAMPUS  
FRIEDBERG**

**MND**

Mathematik, Naturwissenschaften  
und Datenverarbeitung

# Entwicklung verteilter Anwendungen

Dokumentation

## ***Bundesliga-ChatBot***

### **Projektteam**

Kamal Badawi – 5236028

Abdal Ahmad – 5566419

### **Betreuer**

Prof. Dr. Thomas Farrenkopf

Sommersemester 2025

1. Oktober 2025

# Inhaltsverzeichnis

Abbildungsverzeichnis .....	5
1. Einführung .....	6
1.1. Einleitung .....	6
1.2. Nutzung des Chatbots von Null an .....	6
1.2.1. Voraussetzungen .....	6
1.2.2. Backend starten (Python + FastAPI) .....	7
1.2.3. Frontend starten (Node.js + Yarn) .....	8
1.2.4. Nutzung .....	9
2. Architektur .....	9
2.1. Use-Case-Diagramm .....	9
2.2. Blockdiagramm .....	10
2.3. Datenflussdiagramm .....	12
2.4. Zustandsdiagramm .....	15
3. Technologien .....	17
3.1. Frontend .....	17
3.2. Backend .....	17
3.3. Schnittstellen .....	17
3.4. Datenbank I - MongoDB .....	17
3.5. Datenbank II - SQLite3 .....	18
3.6. Entwicklungstools .....	18
4. Technische Detailbeschreibung .....	18
4.1. Zweck .....	18
4.2. Aktueller Ablauf (Ist-Zustand) .....	18
4.3. Fehlerfall / Fallback .....	18
4.4. Geplante Verbesserungen (Soll-Zustand) .....	19
4.4.1. Redundanz der Datenbanken .....	19
4.4.2. Trennung: Admin-Sicht vs. Nutzer-Sicht .....	19
4.4.3. Intervall während Spieltagen .....	19
4.4.4. Caching auf Nutzer-Sicht .....	19
4.5. Weitere technische Empfehlungen .....	20
4.6. Konkrete To-Dos / Checkliste für Entwickler .....	20
5. Endpunktbeschreibung und Ausfallsicherheit .....	21
6. API-Keys erstellen .....	34
6.1. Gmail-Passkey erstellen .....	34
6.2. Kostenlosen Google Gemini 2.5 Pro API-Key erstellen .....	35
6.3. VoiceRSS-API Key erstellen .....	36
6.4. AssemblyAI-API Key erstellen .....	38

6.5 MongoDB Key erstellen .....	39
6.6 OpenLigaDB-API .....	39
7. Anwendungsbeispiel .....	40
8. Test-Szenarien und Teststrategie .....	50
8.1. Frontend Tests (React + TailwindCSS) .....	50
8.1.1 Initiales Laden .....	50
8.1.2 Frage stellen .....	50
8.1.3 Neuer Chat starten .....	50
8.1.4 Chatverlauf öffnen .....	50
8.1.5 Chat durchsuchen/filtern .....	50
8.1.6 Chat löschen (aktiver Chat) .....	50
8.1.7 Chat löschen (nicht-aktiver Chat) .....	51
8.1.8 Export als PDF (lokal speichern) .....	51
8.1.9 Chat per E-Mail senden (Gmail/SendGrid) .....	51
8.1.10 Responsives Layout / Mobile View .....	51
8.2 Backend Tests (FastAPI, LangChain, MongoDB, SQLite) .....	51
8.2.1 API /question .....	51
8.2.2 API /conversations_info .....	51
8.2.3 Speicherung in MongoDB .....	52
8.2.4 SQLite Fallback .....	52
8.2.5 RAG mit Langchain .....	52
8.2.6 PDF-Erstellung Backend .....	52
8.2.7 Mailversand Gmail .....	52
8.2.8 Mailversand SendGrid .....	52
8.3. Datenbank Tests (MongoDB + SQLite) .....	53
8.3.1 MongoDB Chat-Verwaltung .....	53
8.3.2 SQLite OpenLigaDB Datenhaltung .....	53
8.3.3 Konsistenzprüfung MongoDB/SQLite .....	53
8.3.4 Filter-Queries in MongoDB .....	53
8.4. End-to-End (System + User Flows) .....	53
8.4.1 Vollständige Session .....	53
8.4.2 Fehlerhandling (API Key ungültig) .....	53
8.4.3 Fehlerhandling (OpenLigaDB Timeout → SQLite) .....	54
8.4.4 Fehlerhandling (Datenbank nicht erreichbar) .....	54
8.4.5 Löschlogik aktiv/nicht-aktiv .....	54
8.5. Nicht-funktionale Tests .....	54
8.5.1 Performance (100 gleichzeitige User) .....	54
8.5.2 Sicherheit: SQL Injection .....	54

8.5.3 Sicherheit: XSS im Frontend .....	54
8.5.4 Zugriffsschutz conversation_id & user_id.....	55
8.5.5 Netzwerkunterbrechung.....	55
8.5.6 Usability (Mobile/Desktop) .....	55
9. Authentifizierung und Zugriffskontrolle .....	55
10. Versionskontrollsystem (Git + GitHub) .....	56
11. Fazit.....	56
Quellen.....	58

# Abbildungsverzeichnis

Abbildung 1: Use-Case Diagramm - Bundesliga-ChatBot.....	10
Abbildung 2: Blockdiagramm - Bundesliga-ChatBot .....	12
Abbildung 3: Datenflussdiagramm der Systemarchitektur - Bundesliga-ChatBot .....	13
Abbildung 4: Prompt - Bundesliga-ChatBot .....	14
Abbildung 5: SQLite3-Datenbanken .....	15
Abbildung 6: Zustandsdiagramm - Bundesliga-ChatBot.....	16

# 1. Einführung

## 1.1. Einleitung

Diese Dokumentation beschreibt die Entwicklung und Funktionsweise des Bundesliga-ChatBots, einer spezialisierten Anwendung, die Nutzer in Echtzeit mit Informationen und Statistiken zur Fußball-Bundesliga versorgt. Der Bundesliga-ChatBot bezieht seine Daten aus mehreren APIs, komplett über die *OpenLigaDB*-APIs, und nutzt zusätzlich ein leistungsstarkes Large Language Model (LLM) namens *Google Gemini 2.5 Pro*, um kontextbezogene Antworten auf Nutzeranfragen zu generieren.

Die Architektur des Systems kombiniert moderne Frontend- und Backend-Technologien. Das Frontend basiert auf *React* und *TailwindCSS* und stellt die Benutzeroberfläche bereit, über die Nutzer ihre Fragen stellen, Chatverläufe einsehen und exportieren sowie weitere Interaktionen durchführen können. Das Backend wird in *Python* mit *FastAPI* umgesetzt und übernimmt die zentrale Steuerung: Es empfängt Anfragen vom Frontend, kommuniziert mit den Datenbanken und APIs, erstellt Chat-Prompts für das Large Language Model (LLM) und gibt die generierten Antworten zurück.

Zur Speicherung und Verarbeitung der Daten kommen mehrere Technologien zum Einsatz. *MongoDB* dient als Hauptdatenbank für Chat-Konversationen, Dialog-Items und Metadaten, während *SQLite* die Daten aus der *OpenLigaDB*-API persistiert und als Fallback bei Ausfällen der API dient. Zusätzlich wird eine Vektordatenbank (*Langchain*) verwendet, um relevante Embeddings für Retrieval Augmented Generation (RAG) bereitzustellen, wodurch das LLM in die Lage versetzt wird, Antworten mit zusätzlichem Kontext zu generieren.

Die Dokumentation deckt alle relevanten Aspekte des ChatBots ab: Sie beschreibt die Installation und Nutzung von Frontend und Backend, die Einrichtung notwendiger API-Keys, die technische Architektur, die eingesetzten Datenbanken und Vektorsysteme, die Funktionsweise der Endpunkte sowie die Datenflüsse zwischen den Komponenten. Abschließend wird die Anwendung in einem Anwendungsbeispiel vorgestellt, die die Echtzeit-Kommunikation, Datenabfrage und Ausgabe von Antworten demonstriert.

## 1.2. Nutzung des Chatbots von Null an

In diesem Abschnitt wird Schritt für Schritt erklärt, wie der Bundesliga-ChatBot von der Installation bis zur Nutzung eingerichtet werden kann. Zunächst werden die technischen Voraussetzungen beschrieben, die für das Backend und Frontend notwendig sind. Anschließend wird das Starten des Backends (*Python* + *FastAPI*) und des Frontends (*Node.js* + *Yarn*) erläutert. Abschließend wird gezeigt, wie der Bundesliga-ChatBot im Browser aufgerufen und genutzt werden kann.

Die Anleitung richtet sich an Einsteiger und setzt keine Vorkenntnisse voraus, außer der Grundkenntnis im Umgang mit Kommandozeile/Terminal. Durch die klar strukturierten Schritte können Sie den Bundesliga-ChatBot lokal starten, Nachrichten eingeben und sofort Antworten in Echtzeit erhalten.

### 1.2.1. Voraussetzungen

- *Python* (Version 3.11.3)
- *Node.js* und *Yarn*

- *MongoDB* (lokal oder Cloud, z. B. *MongoDB Atlas*)
- API-Keys für die `.env`-Datei im Backend (siehe Kapitel „API-Keys erstellen“)

### 1.2.2. Backend starten (*Python + FastAPI*)

1. Projekt als `zip`-Datei herunterladen und ins Backend (in VS Code) wechseln:  
`cd backend`

```
PS C:\Users\abahm\OneDrive\Desktop\Bundesliga-ChatBot-Gemini-master> cd backend
```

2. Virtuelle Umgebung erstellen und aktivieren:  
`python -m venv .venv`  
`.venv/Scripts/activate` (Windows)
3. *UV* installieren (schnellerer Paketmanager):  
`pip install uv`

```
(.venv) PS C:\Users\abahm\OneDrive\Desktop\Bundesliga-ChatBot-Gemini\backend> pip install uv
Collecting uv
  Downloading uv-0.8.12-py3-none-win_amd64.whl (20.3 MB)
    20.3/20.3 MB 6.1 MB/s eta 0:00:00
Installing collected packages: uv
Successfully installed uv-0.8.12
```

4. Abhängigkeiten installieren:  
`uv pip install -r requirements.txt`

```
(.venv) PS C:\Users\abahm\OneDrive\Desktop\Bundesliga-ChatBot-Gemini\backend> uv pip install -r requirements.txt
Resolved 105 packages in 6.01s
Prepared 25 packages in 2m 18s
Uninstalled 1 package in 4.17s
Installed 105 packages in 10m 21s
+ aiohappyeyeballs==2.6.1
```

5. Umgebungsvariablen in `.env`-Datei setzen:  
*Siehe Kapitel API-Keys erstellen*
6. Backend starten:  
`uvicorn main:app --reload`

### 1.2.3. Frontend starten (Node.js + Yarn)

1. Neues Terminal öffnen und ins Frontend wechseln:  
`cd frontend`
2. Abhängigkeiten (yarn) installieren:  
`yarn install`

```
PS C:\Users\abahm\OneDrive\Desktop\Bundesliga-ChatBo
● t-Gemini> cd frontend
PS C:\Users\abahm\OneDrive\Desktop\Bundesliga-ChatBo
● t-Gemini\frontend> yarn install
yarn install v1.22.22
warning package-lock.json found. Your project contains lock files generated by tools other than Yarn. It is advised not to mix package managers in order to avoid resolution inconsistencies caused by unsynchronized lock files. To clear this warning, remove package-lock.json.
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...
Done in 734.60s.
PS C:\Users\abahm\OneDrive\Desktop\Bundesliga-ChatBo
● t-Gemini\frontend> & C:/Users/abahm/OneDrive/Desktop/Bundesliga-ChatBot-Gemini/backend/.venv/Scripts/Activate.ps1
○ (.venv) PS C:\Users\abahm\OneDrive\Desktop\Bundeslig
```

3. Frontend starten:  
`yarn dev`



```
(.venv) PS C:\Users\abahm\OneDrive\Desktop\Bundesliga-ChatBot-Gemini\frontend> yarn dev
yarn run v1.22.22
$ vite

VITE v4.5.14 ready in 3354 ms

→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h to show help

█
```

### 1.2.4. Nutzung

Um die Anwendung zu verwenden, öffnen Sie zunächst den lokalen Frontend-Link im Browser. In diesem Beispiel lautet die URL: <http://localhost:5173/>. Sobald die Seite geladen ist, können Sie im vorgesehenen Eingabefeld Chatnachrichten eingeben. Der Chatbot verarbeitet Ihre Eingaben sofort und liefert die Antworten in Echtzeit. Eine detaillierte Demonstration der Funktionsweise finden Sie im Kapitel „Anwendungsbeispiel“.

## 2. Architektur

### 2.1. Use-Case-Diagramm

Das folgende Use-Case-Diagramm beschreibt die verschiedenen Interaktionsmöglichkeiten der Nutzer mit dem Bundesliga-ChatBot und unterscheidet zwischen einem allgemeinen Nutzer und einem Nutzer mit Behinderung, der zusätzliche barrierefreie Funktionen verwenden kann. Der Nutzer kann beispielsweise Fragen zu Spieltagen, Ergebnissen oder anderen relevanten Spieldaten stellen, woraufhin das LLM passende Antworten aus den vorhandenen Datenquellen generiert. Zudem besteht die Möglichkeit, den Chatverlauf einzusehen, wobei durch eine Filter- und Suchfunktion bestimmte Inhalte gezielt gefunden werden können. Der Nutzer kann den gespeicherten Verlauf außerdem vollständig löschen oder exportieren; der Export umfasst dabei sowohl die Erstellung einer PDF-Datei als auch den Versand der Konversation per E-Mail. Antworten des ChatBots können direkt kopiert oder akustisch vorgelesen werden, sodass die Informationen flexibel genutzt werden können. Für Nutzer mit Behinderung sind erweiterte Funktionen vorgesehen, wie etwa die Möglichkeit, Fragen per Spracheingabe zu stellen, die in einer späteren Version über *AssemblyAI* (Speech-to-Text) unterstützt wird. Alternativ können Antworten auch als Sprachausgabe wiedergegeben werden, was zukünftig über *VoiceRSS* (Text-to-Speech) realisiert werden soll. Darüber hinaus können Nutzer Echtzeit-Updates direkt aus dem Internet abrufen, um stets aktuelle Spieldaten zu erhalten.

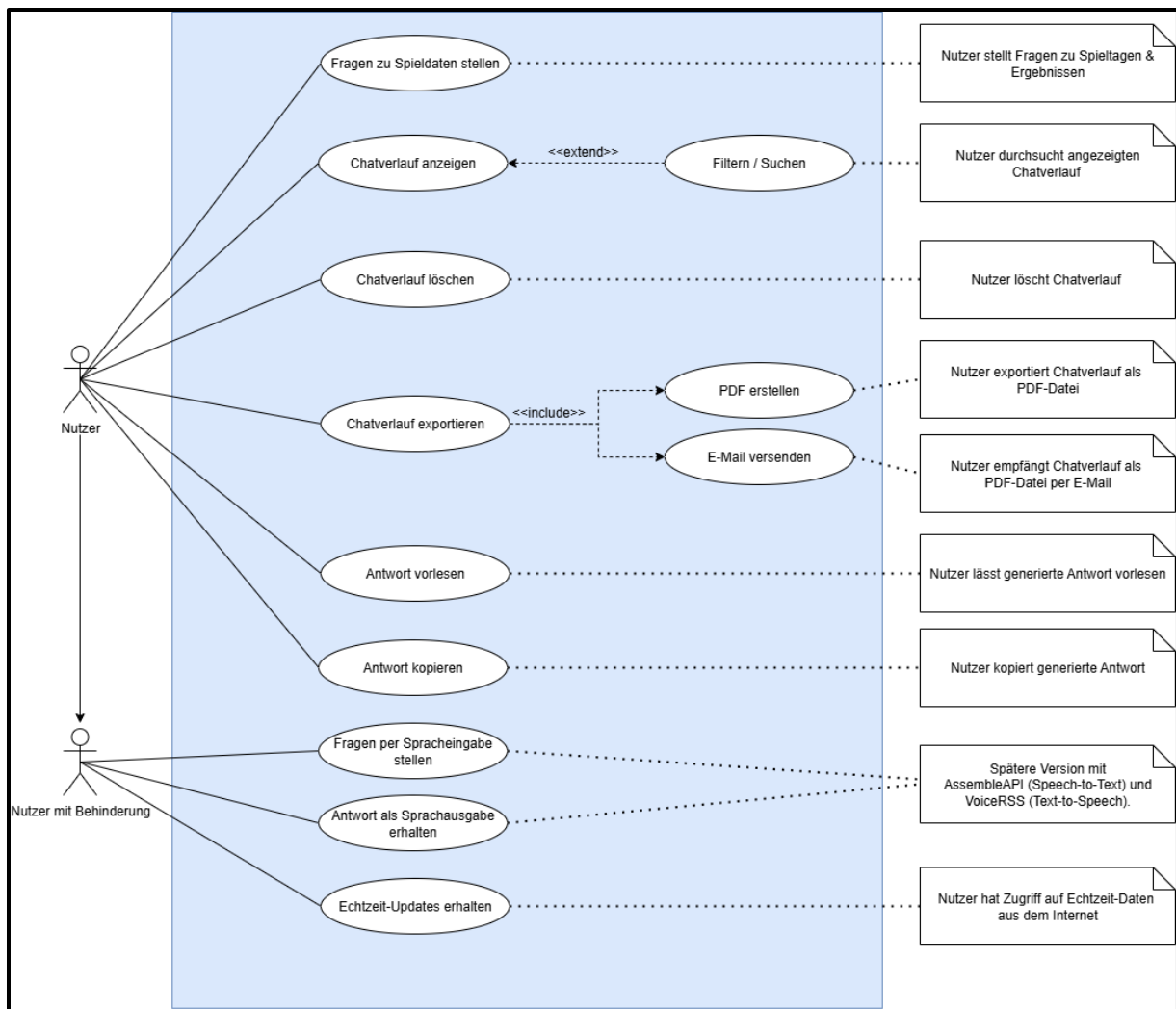


Abbildung 1: Use-Case-Diagramm - Bundesliga-ChatBot

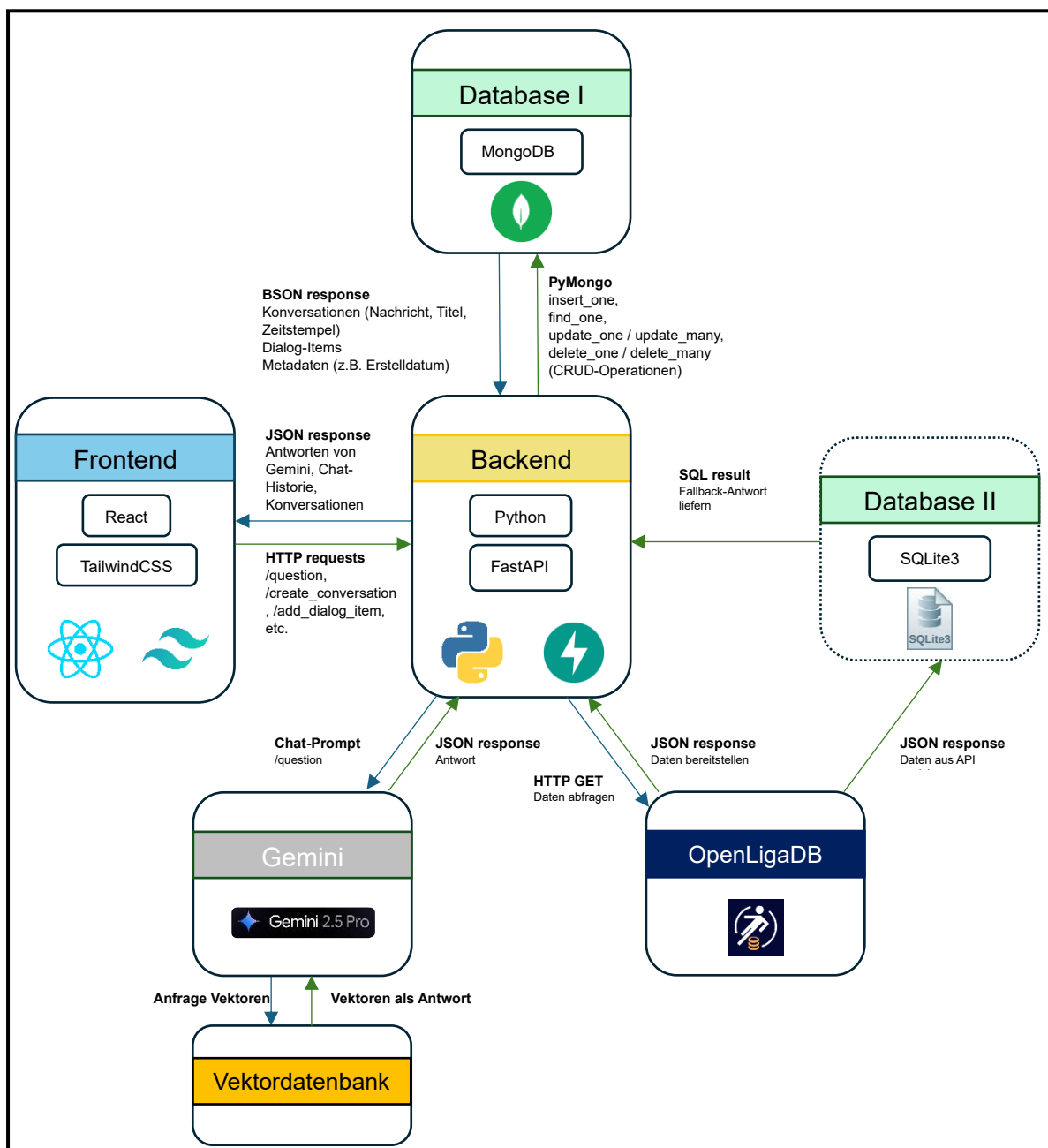
## 2.2. Blockdiagramm

Das folgende Blockdiagramm veranschaulicht die Architektur des Bundesliga-ChatBots sowie den Datenfluss zwischen den einzelnen Komponenten. Das Frontend, umgesetzt mit *React* und *TailwindCSS*, dient als Benutzeroberfläche, über die Nutzer ihre Anfragen stellen. Diese Anfragen werden per *HTTP*-Requests - beispielsweise über die Endpunkte `/question`, `/create_conversation` oder `/add_dialog_item` - an das Backend gesendet. Die vom Backend generierten Antworten, wie die Ausgaben des LLM *Gemini 2.5 Pro* oder gespeicherte Chat-Historien, werden als *JSON*-Response zurückgegeben und in Echtzeit im Frontend angezeigt.

Das Backend, realisiert mit *Python* und *FastAPI*, übernimmt die zentrale Steuerung. Es empfängt Anfragen vom Frontend und koordiniert die Kommunikation mit den Datenbanken, externen APIs und dem LLM. Dabei greift es über *PyMongo* auf Database I (*MongoDB*) zu, um Konversationen, Dialog-Items und Metadaten wie Zeitstempel zu speichern und zu verwalten. Für aktuelle Spieldaten nutzt das Backend die *OpenLigaDB*-API, die per *HTTP* GET abgefragt wird. Die erhaltenen Daten werden in Database II (*SQLite3*) persistiert, sodass im Falle eines API-Ausfalls auf die lokal gespeicherten Daten zurückgegriffen werden kann. Über den Endpunkt `/question` erstellt das Backend einen Chat-Prompt, der an das LLM gesendet wird.

Database I (*MongoDB*) fungiert als Hauptdatenbank für die Chat-Konversationen und speichert Nachrichten, Titel, Zeitstempel und weitere Metadaten. Das Backend erhält von MongoDB strukturierte *JSON*-Responses, etwa komplette Gesprächsverläufe oder einzelne Konversationseinträge. Database II (*SQLite3*) enthält die Daten der OpenLigaDB-API und stellt per Fallback die zuletzt gespeicherten Informationen bereit, wenn die API nicht erreichbar ist.

Das LLM *Gemini 2.5 Pro* generiert die Antworten auf Nutzeranfragen. Das Backend sendet Chat-Prompts an das LLM, das diese verarbeitet und die Antworten als *JSON*-Response zurückliefert. Um kontextbasierte Antworten zu ermöglichen, werden die Anfragen zusätzlich mit relevanten Vektoren aus der Vektordatenbank angereichert. Diese Embeddings liefern dem LLM zusätzlichen Kontext im Rahmen eines *Langchain*-RAG-Ansatzes (Retrieval-Augmented Generation), sodass die Antworten präziser auf die Nutzeranfragen abgestimmt sind. Die generierten Antworten werden schließlich an das Frontend weitergeleitet und dort angezeigt.



## 2.3. Datenflussdiagramm

Die folgende Abbildung zeigt die Softwarearchitektur des Bundesliga-Chatbots, die in mehrere Schichten unterteilt ist: Frontend, Backend, Datenquellen (APIs), *Google Gemini 2.5 Pro*-LLM und *Langchain*-RAG.

Das Frontend bildet die Schnittstelle für den Nutzer, der beispielsweise eine Frage wie „Eintracht Frankfurt letzter Spieltag?“ stellt und daraufhin eine Antwort erhält, etwa „Frankfurt spielte am 16. Spieltag gegen FC St. Pauli.“. Die Kommunikation mit dem Backend erfolgt über *HTTP*-Requests.

Das Backend wurde als *FastAPI*-Service in *Python* realisiert. Es nimmt die Nutzeranfrage über den Endpunkt `/question` entgegen, bereitet die Daten für das Large Language Model *Google Gemini 2.5 Pro* auf und steuert die Datenflüsse zwischen den APIs, dem RAG-System und dem LLM.

Das LLM verarbeitet die Frage zusammen mit einem Dictionary aller DataFrames sowie den Beschreibungen der einzelnen Datenquellen und generiert daraus die passende Antwort.

Ergänzend kommt das Verfahren der Retrieval-Augmented Generation (RAG) zum Einsatz. Dabei wird das LLM nicht mit allen verfügbaren Daten auf einmal versorgt, sondern ausschließlich mit den für die jeweilige Frage relevanten Informationen.

Zu diesem Zweck werden die Daten in Vektoren umgewandelt und in einer Vektordatenbank abgelegt. Stellt der Nutzer eine Anfrage, durchsucht das System diese Datenbank nach den bestpassenden Informationen und übergibt nur diese Teilmenge an das LLM. Auf diese Weise reduziert sich die Menge an Input, was zu schnelleren Antworten, geringeren Kosten und gleichzeitig präziseren Ergebnissen führt, da ausschließlich relevante Daten berücksichtigt werden. In der Architektur des Bundesliga-Chatbots trägt RAG somit wesentlich dazu bei, den Rechenaufwand zu optimieren und eine zielgerichtete Beantwortung der Nutzerfragen sicherzustellen.

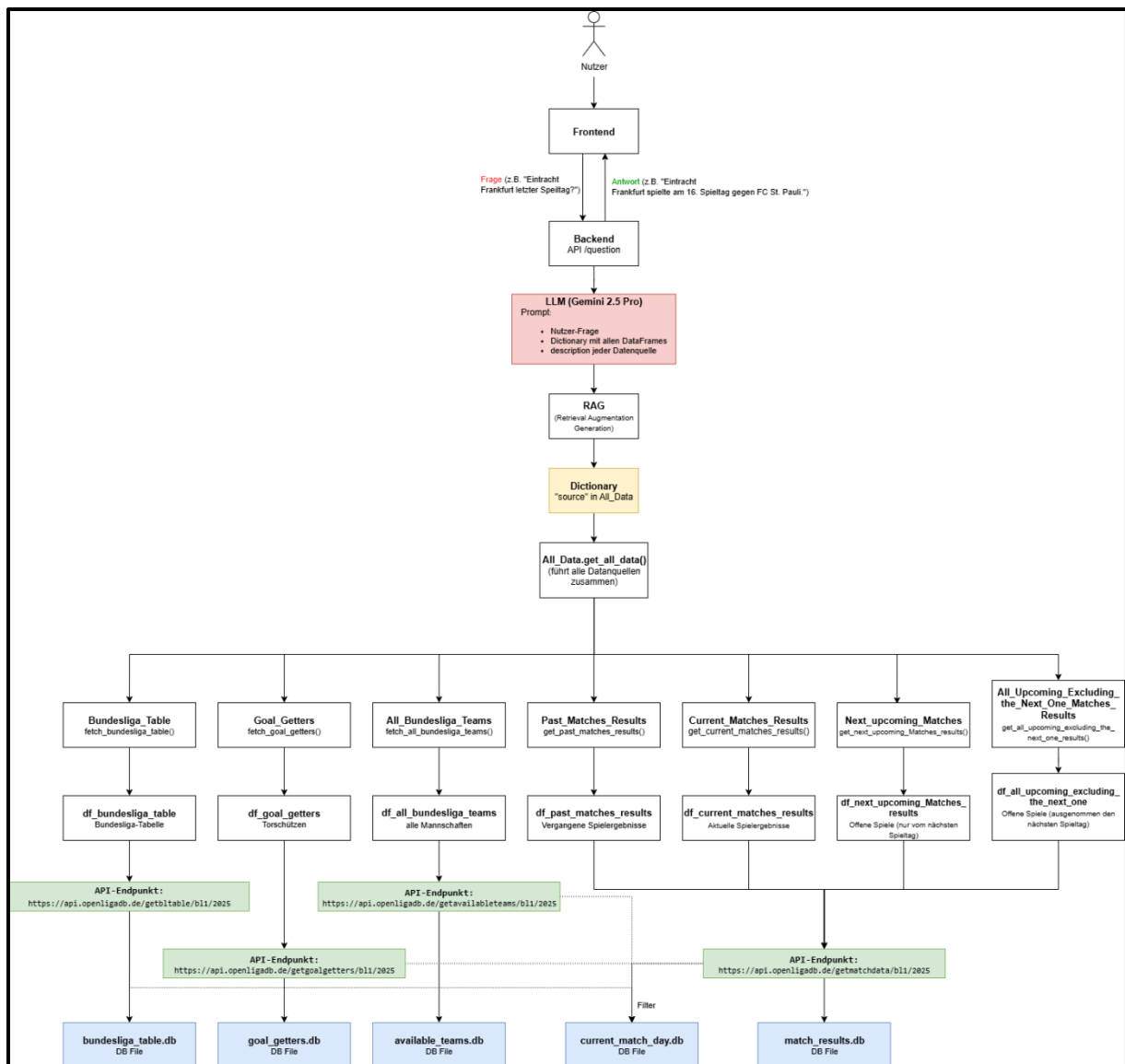


Abbildung 3: Datenflussdiagramm der Systemarchitektur - Bundesliga-ChatBot

Um die Nutzerfragen präzise mit den passenden Datenquellen zu verknüpfen, erhält das LLM einen speziell formulierten Prompt, der neben der ursprünglichen Frage auch strukturierte Informationen über die verfügbaren Datenquellen umfasst.

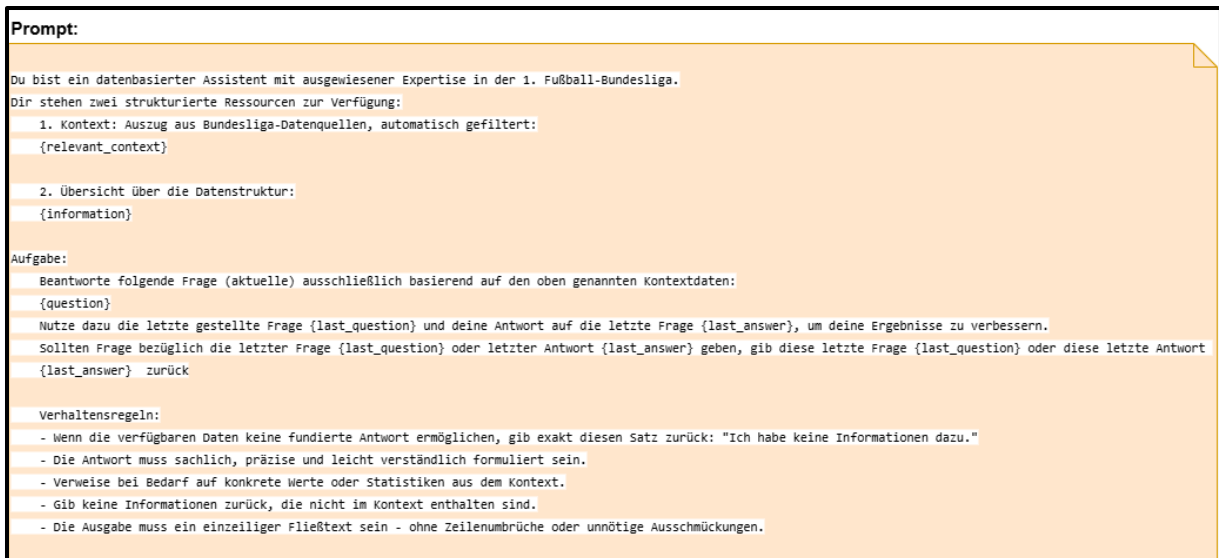


Abbildung 4: Prompt - Bundesliga-ChatBot

Das Dictionary `All_Data.get_all_data` fungiert als zentraler Sammelpunkt für alle Datenquellen des Bundesliga-ChatBots. Es enthält unter anderem den Tabellenstand, Informationen zu Mannschaften, Torschützen sowie die Ergebnisse vergangener, aktueller und kommender Spiele. Diese konsolidierte Datenbasis wird anschließend an das RAG-System und das LLM übergeben, um kontextbasierte Antworten zu generieren.

Die Daten stammen aus der *OpenLigaDB*-API und werden in *SQLite3*-Datenbanktabellen (siehe Abbildung 5) gespeichert, wodurch ein Fallback-Sicherheitsmechanismus gewährleistet ist. Jede Datenquelle besitzt eine eigene Methode sowie eine spezifische *DataFrame*-Struktur. Die Methode `fetch_bundesliga_table` ruft beispielsweise die Tabelleninformationen über den API-Endpunkt <https://api.openligadb.de/getbltable/bl1/2025> ab, inklusive Tabellenplatz, Punkte, Tore, Siege, Niederlagen und weiterer Kennzahlen, und speichert diese in der Datenbank `bundesliga_table.db`. Die Methode `fetch_goal_getters` liefert die Torschützen inklusive Spielernamen und Anzahl der Tore und speichert diese Daten in `goal_getters.db`. `fetch_all_bundesliga_teams` ruft eine Liste aller Teams ab und speichert diese in `available_teams.db`.

Ergebnisse vergangener Spieltage werden über `get_past_matches_results`, aktuelle Spieltage über `get_current_matches_results` und die kommenden Spiele des nächsten Spieltags über `get_next_upcoming_matches_results` bereitgestellt. Für alle zukünftigen Spieltage außer dem nächsten wird `get_all_upcoming_excluding_the_next_one_results` verwendet.

Alle diese Methoden greifen intern auf `All_Matches_Results` zurück, das den API-Endpunkt <https://api.openligadb.de/getmatchdata/bl1/2025> nutzt. Dabei erfolgt eine passende Filterung über die API (<https://api.openligadb.de/getcurrentgroup/bl1>), um zwischen vergangenen, aktuellen und zukünftigen Spielen zu unterscheiden. Die Daten werden in den Datenbanken `current_match_day.db` (Filter) sowie `matches_results.db` gespeichert. Zusätzlich werden Team-Icons in `teams_icons_urls.db` gesichert, um sie gegebenenfalls in der Zukunft für die Darstellung oder weitere Funktionen nutzen zu können.







 available_teams.db	29.09.2025 10:29	Data Base File	12 KB
 bundesliga_table.db	29.09.2025 10:29	Data Base File	12 KB
 current_match_day.db	29.09.2025 10:29	Data Base File	12 KB
 goal_getters.db	29.09.2025 10:29	Data Base File	12 KB
 match_results.db	29.09.2025 10:29	Data Base File	36 KB
 teams_icons_urls.db	21.09.2025 20:57	Data Base File	12 KB

Abbildung 5: SQLite3-Datenbanken

## 2.4. Zustandsdiagramm

Der Ablauf beginnt mit einem Initialzustand, der den Start einer neuen Konversation markiert. Im ersten Schritt wird eine neue Konversation eröffnet, woraufhin eine Frage an das System gestellt wird, die über den Endpunkt `/question` ausgelöst wird. Diese Anfrage wird anschließend an das Large Language Model *Gemini 2.5 Pro* weitergeleitet, das den Prompt verarbeitet und einen Antwortvorschlag zurückliefert. Im nächsten Schritt erfolgt die Antwortgenerierung im Backend, bei der die vom LLM erzeugte Antwort gegebenenfalls mit zusätzlichen Informationen - beispielsweise aus der *OpenLigaDB*-API - angereichert wird, um eine vollständige Antwort zu erzeugen. Ist die API verfügbar, werden aktuelle Daten wie Spielergebnisse eingebunden; sollte sie ausfallen, greift das System auf den *SQLite*-Fallback zurück und verwendet dort gespeicherte Daten. Beide Verarbeitungswege münden im Zustand „Antwort erhalten“, sodass das Ergebnis vorliegt. Anschließend wird die Antwort in der *MongoDB* gespeichert, um den Verlauf der Konversation dauerhaft festzuhalten. Danach hat der Nutzer die Möglichkeit, zwischen drei Aktionen zu wählen: den Download der Konversation als PDF, den Versand per E-Mail oder das Löschen der Konversation. Unabhängig von der gewählten Aktion endet der Ablauf schließlich im finalen Zustand, der das Beenden des Prozesses kennzeichnet.

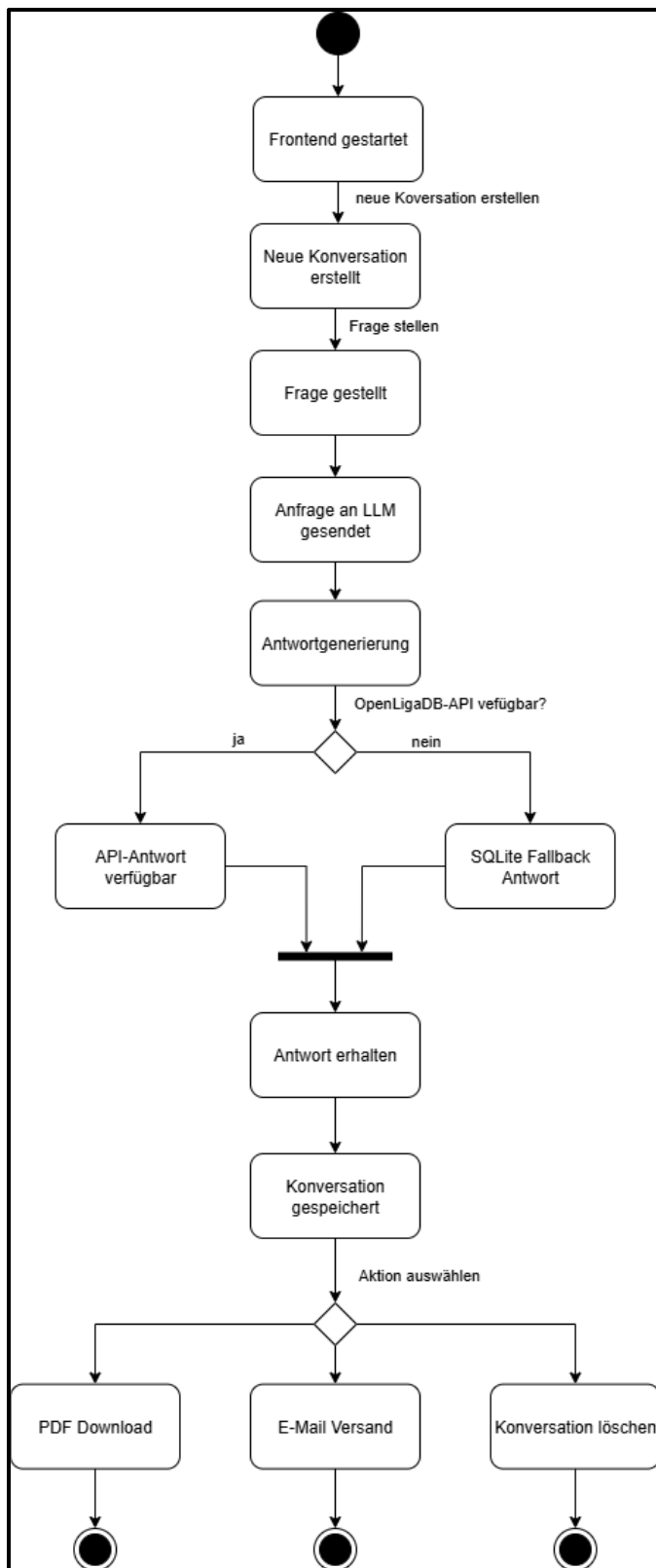


Abbildung 6: Zustandsdiagramm - Bundesliga-ChatBot



## 3. Technologien

### 3.1. Frontend

Das Frontend wurde mit dem *JavaScript*-Framework *React* umgesetzt. *React* eignet sich besonders gut, da es eine komponentenbasierte Architektur bietet, die eine modulare und wiederverwendbare Entwicklung ermöglicht. Dadurch lässt sich das Chat-Interface effizient umsetzen und in Echtzeit aktualisieren.

Zur Gestaltung der Benutzeroberfläche wird *TailwindCSS* eingesetzt. *TailwindCSS* ermöglicht durch Utility-Klassen die schnelle Entwicklung eines modernen, responsiven Designs und reduziert den Aufwand für individuelle CSS-Regeln.

Die Kommunikation mit dem Backend erfolgt über *HTTP*-Requests und *JSON*-Responses. *JSON* ist ein leichtgewichtiges Datenformat, das von *JavaScript* (Frontend) und *Python* (Backend) nativ unterstützt wird und so den Austausch von Nutzerfragen, Antworten und Chat-Historien effizient macht.

### 3.2. Backend

Das Backend basiert auf *Python* und wurde mit *FastAPI* umgesetzt. *Python* ist eine etablierte Sprache für Datenverarbeitung und KI-Integration und ermöglicht eine schnelle Implementierung. *FastAPI* bietet eine hohe Performance und erleichtert die Entwicklung durch automatische API-Dokumentation (*Swagger-UI*).

Zur Bereitstellung der KI-Antworten wird *Gemini 2.5 Pro* angebunden. Dieses Large Language Model liefert besonders schnelle Antworten bei gleichzeitig hoher Genauigkeit und eignet sich daher optimal für den Einsatz in einem interaktiven Chatbot.

### 3.3. Schnittstellen

Die Kommunikation zwischen Frontend und Backend erfolgt über eine *REST*-API, die strukturierte *JSON*-Daten austauscht.

*REST* wurde gewählt, da es leichtgewichtig und standardisiert ist und sich problemlos mit Webanwendungen wie *React* verbinden lässt. *JSON* wiederum ist ideal, da es sowohl in *JavaScript* (Frontend) als auch in *Python* (Backend) unterstützt wird und damit einen effizienten Datenaustausch ohne zusätzliche Konvertierung ermöglicht.

### 3.4. Datenbank I - *MongoDB*

Für die Speicherung der Konversationen wird *MongoDB* eingesetzt. Die dokumentenorientierte *NoSQL*-Datenbank speichert Daten flexibel im *JSON*-ähnlichen *BSON*-Format, wodurch sich Chat-Verläufe mit variabler Struktur effizient verwalten lassen.

Die Anbindung an das Backend erfolgt über *PyMongo*. Die offizielle *Python*-Bibliothek für *MongoDB* ermöglicht den Zugriff auf zentrale Methoden wie `insert_one` (neue Konversation anlegen), `find/find_one` (Konversationen abrufen), `update_one/update_many` (Dialog erweitern) und `delete_one/delete_many` (Konversationen löschen).

### 3.5. Datenbank II - *SQLite3*

Für die Speicherung der aus der *OpenLigaDB*-API abgerufenen Daten wird *SQLite3* eingesetzt. Hauptgrund für die Wahl ist die Möglichkeit, auch bei Ausfall der API weiterhin auf die zuletzt gespeicherten Daten zugreifen zu können. Damit stellt *SQLite* ein zentrales Fallback dar, das die Verfügbarkeit des ChatBots sicherstellt. Gleichzeitig ist *SQLite* leichtgewichtig, benötigt keinen separaten Server und lässt sich direkt aus *Python* ansprechen, was die Integration besonders einfach und ressourcenschonend macht.

### 3.6. Entwicklungstools

Für die Entwicklung wurden unterschiedliche Werkzeuge eingesetzt, die jeweils passgenau auf die Anforderungen der einzelnen Komponenten zugeschnitten sind. Zur Installation von *Python*-Bibliotheken kommt *uv* zum Einsatz, das durch parallele und besonders schnelle Prozesse eine effiziente Einrichtung der Entwicklungsumgebungen ermöglicht - ein wesentlicher Vorteil bei Projekten mit zahlreichen Abhängigkeiten. Im Frontend wird *yarn* als Paketmanager für *React* verwendet. Die Wahl fiel bewusst auf *yarn* anstelle von *npm*, da es durch optimierte Caching-Mechanismen eine schnellere, konsistentere Verwaltung von Abhängigkeiten bietet und damit die Stabilität des Build-Prozesses erhöht. Für das Backend fungiert *Uvicorn* als ASGI-Server für *FastAPI*. Dank seiner Leichtgewichtigkeit, hohen Performance und asynchronen Architektur erlaubt er die parallele Verarbeitung zahlreicher Anfragen und stellt so die notwendige Skalierbarkeit für einen Chatbot mit potenziell hoher Nutzerlast sicher.

## 4. Technische Detailbeschreibung

### 4.1. Zweck

Der Bundesliga-ChatBot aggregiert Live- und Statistikinformationen aus mehreren *OpenLigaDB*-APIs, speichert sie lokal und stellt auf dieser Basis Antworten an Nutzeranfragen bereit. Ziel ist hohe Verfügbarkeit (auch bei Ausfall einzelner APIs), schnelle Antwortzeiten und eine saubere Trennung zwischen Daten-Ingestion (Admin/Backend) und Anfrage-Beantwortung (Nutzer-Sicht).

### 4.2. Aktueller Ablauf (Ist-Zustand)

Bei jeder Nutzeranfrage werden die *OpenLiga*-APIs abgefragt, bei Bedarf *SQLite*-Tabellen angelegt oder aktualisiert und zudem Vektorindizes in der Vektordatenbank erstellt oder neu aufgebaut. Da Ingestion, Persistenz und Index-Erstellung hierbei synchron im Rahmen der Anfrage ausgeführt werden, führt dieses Vorgehen zu deutlich erhöhten Latenzzeiten.

### 4.3. Fehlerfall / Fallback

Wenn eine API ausfällt, verwendet der Chatbot den zuletzt erfolgreichen Datenstand aus den lokalen *SQLite*-Tabellen. Dadurch bleiben Antworten möglich, auch wenn Live-Daten fehlen.

## 4.4. Geplante Verbesserungen (Soll-Zustand)

### 4.4.1. Redundanz der Datenbanken

Zukünftig ist vorgesehen, pro Datenbank zwei Duplikate vorzuhalten, sodass insgesamt drei Kopien jedes Datenbestands existieren. Auf diese Weise soll der Ausfall einer physischen Datei oder Instanz durch ein automatisches Umschalten auf eine redundante Kopie abgefangen werden. Dabei ist jedoch zu berücksichtigen, dass *SQLite* als dateibasierte Lösung nur eingeschränkt für verteilte Replikation und hohen parallelen Schreibzugriff geeignet ist. Für echte Hochverfügbarkeit empfiehlt es sich daher, den Einsatz einer Client-Server-Datenbank wie beispielsweise *PostgreSQL* oder alternativ ein geeignetes Replikationsverfahren zu evaluieren.

### 4.4.2. Trennung: Admin-Sicht vs. Nutzer-Sicht

In der zukünftigen Architektur wird zwischen einer Admin-Sicht und einer Nutzer-Sicht unterschieden.

Aus Admin-Sicht steht eine Oberfläche beziehungsweise ein Service zur Verfügung, über den Datenpipelines konfiguriert, gestartet, gestoppt und überwacht werden können. Diese Pipelines laufen in einer isolierten Umgebung (z. B. in separaten Prozessen oder Containern) und übernehmen folgende Aufgaben: die periodische Abfrage der *OpenLigaDB*-APIs, die Validierung und Normalisierung der Daten, atomare Persistenz-Updates in den Datenbanken, die Berechnung und Vorverarbeitung von Embeddings sowie die inkrementelle (wenn möglich) Aktualisierung der Vektorindizes.

Aus Nutzer-Sicht erfolgen ausschließlich lesende Abfragen gegen die bereits vorbereiteten Daten und Indizes. Eine direkte Ingestion von API-Daten durch den Nutzer findet nicht statt.

### 4.4.3. Intervall während Spieltagen

Während aktiver Spieltage sollen die Datenpipelines in einem Intervall von drei Sekunden aktualisiert werden, um nahezu in Echtzeit auf neue Spielereignisse reagieren zu können. Nach Abschluss des Spieltags wird diese Hochfrequenz-Pipeline automatisch beendet, da fortlaufende Live-Updates nicht mehr erforderlich sind. Als Empfehlung bietet es sich an, die Steuerung der Pipeline an Spielplänen oder definierten Spielzeiten auszurichten, beispielsweise über einen Scheduler oder den Einsatz externer Metadaten, sodass die Pipeline bedarfsgerecht aktiviert und deaktiviert werden kann.

### 4.4.4. Caching auf Nutzer-Sicht

Zukünftig ist die Einführung eines Caches auf der Nutzer-Sicht vorgesehen, beispielsweise durch den Einsatz von *Redis* oder eines in-memory Caches mit *LRU*-Strategie. Ziel ist es, wiederholte, ressourcenintensive Abfragen zu vermeiden und gleichzeitig die Latenz für die Nutzer deutlich zu reduzieren. Geplant sind verschiedene Caching-Strategien wie endpoint-basierte Caches mit definierter *TTL*, *Query-Result-Caching* sowie eine Invalidierung über Pub/Sub-Events, die von der Admin-Pipeline ausgelöst werden. Ergänzend sollen bereits berechnete Embeddings oder Vektor-Schnappschüsse vorgehalten werden, um den Neuaufbau der Vektorindizes bei jeder Anfrage zu vermeiden und so die Performance weiter zu optimieren.

## 4.5. Weitere technische Empfehlungen

Zukünftig sollen die Abläufe im System stärker auf Atomizität, Performance und Ausfallsicherheit ausgelegt werden. Atomare Updates werden umgesetzt, indem neue Daten zunächst in temporäre Tabellen geschrieben und erst nach vollständiger Verarbeitung entweder atomar umbenannt oder über Transaktionen in die produktiven Tabellen übernommen werden, sodass Nutzer niemals halbfertige Daten sehen.

Beim Vektordatenbank-Handling sind inkrementelle Updates (Upserts) vorgesehen, um vollständige Rebuilds zu vermeiden. Zusätzlich werden vorab berechnete Embeddings in persistenter Form - beispielsweise als Datei oder im Cache - gespeichert, um den Rechenaufwand bei Nutzeranfragen zu minimieren.

Für Fehler- und Ausfallmanagement kommen Circuit-Breaker und Backoff-Strategien beim Zugriff auf externe APIs zum Einsatz, ergänzt durch umfassendes Monitoring und Alerts über Health-Checks und Metriken. Eine automatisierte Failover-Logik sorgt dafür, dass im Falle eines Datenbank-Ausfalls nahtlos auf redundante Duplikate umgeschaltet werden kann.

Performance und Skalierbarkeit werden erreicht, indem die Admin-Pipelines in isolierten Containern oder Jobs (z. B. *Kubernetes*-Jobs oder Workflow-Orchestratoren) betrieben werden. Nutzer-Anfragen werden über skalierbare Services verarbeitet, etwa durch mehrere API-Replicas und Load-Balancing.

Schließlich wird die Sicherheit und Datenkonsistenz durch eine strikte Trennung der Zugriffsrechte zwischen Admin- und Nutzerfunktionen gewährleistet. Regelmäßige Backups und Integritätsprüfungen der Datenbank-Duplikate, ergänzt durch Logging und Audit-Trails für alle Datenupdates, sichern die Nachvollziehbarkeit und Stabilität des Systems.

## 4.6. Konkrete To-Dos / Checkliste für Entwickler

Kurzfristig sollen die Abläufe so angepasst werden, dass Ingestion und Indexierung in Hintergrundjobs ausgelagert werden (z. B. mittels *Celery*, *RQ* oder *Kubernetes*-Jobs). Zusätzlich wird ein Cache (Redis) für häufige Nutzeranfragen eingeführt, und Atomic Writes werden umgesetzt, indem Daten zunächst in temporäre Tabellen geschrieben und anschließend per *Swap* oder Transaktionen übernommen werden.

Mittelfristig ist der Aufbau einer Admin-Sicht geplant - entweder als UI oder API -, über die die Pipeline gesteuert, gestartet, gestoppt und der Status überwacht werden kann. Außerdem sollen Mechanismen wie Pub/Sub oder Webhooks implementiert werden, um die Cache-Invalidation von Admin zu Nutzer automatisiert auszulösen, und die Vektorindex-Verarbeitung soll von vollständigen Rebuilds auf inkrementelle Upserts umgestellt werden.

Langfristig wird die Datenbank-Redundanz erweitert, indem pro Datenbank zwei Duplikate vorgehalten werden; hierfür sind entweder Replikationsmechanismen oder ein Umstieg auf eine verteilte Datenbank zu prüfen. Zudem sollte bewertet werden, ob *SQLite* weiterhin ausreicht, oder ob ein DBMS mit nativer Replikation, wie *PostgreSQL*, sinnvoll ist. Abschließend ist die Einführung eines Monitoring- und Alert-Systems (z. B. *Prometheus* und *Grafana*) vorgesehen, um die Systemstabilität und Verfügbarkeit zu gewährleisten.

Zusammenfassend lässt sich festhalten, dass der Bundesliga-ChatBot aktuell seine Informationen aus mehreren *OpenLigaDB*-APIs bezieht. Für jede API wird beim Eintreffen von Daten eine *SQLite*-Tabelle erstellt und mit den entsprechenden Informationen gefüllt. Fällt eine API aus, nutzt der Chatbot den zuletzt verfügbaren Datenstand aus den lokalen Datenbanken.

Zukünftig ist geplant, pro Datenbank zwei Duplikate vorzuhalten, um Ausfälle einzelner Datenbanken abzufangen und die Ausfallsicherheit zu erhöhen.

Derzeit existiert lediglich die Nutzer-Sicht, bei der jede Anfrage die APIs abfragt, die Datenbanken lädt und die Vektor-Datenbanken neu erstellt, was die Antwortzeiten verlängert. Perspektivisch soll eine Admin-Sicht eingeführt werden, über die eine Datenpipeline die Schritte automatisch und isoliert ausführt. Während der Bundesliga-Spieltage werden die Daten im Drei-Sekunden-Takt aktualisiert und weiterverarbeitet; nach Spielende wird die Pipeline gestoppt, da keine Änderungen mehr auftreten.

Ergänzend soll auf der Nutzer-Sicht-Ebene ein Cache implementiert werden, um die Datenbereitstellung zu beschleunigen und die Reaktionszeiten für die Nutzer deutlich zu verbessern.

## 5. Endpunktbeschreibung und Ausfallsicherheit

In diesem Abschnitt werden alle API-Endpunkte des Bundesliga-Chatbots detailliert beschrieben. Für jeden Endpunkt werden Funktion, erwartete Anfragen (Request-Beispiele) und mögliche Ausfälle erläutert. Zusätzlich werden Maßnahmen vorgeschlagen, um den Betrieb auch bei temporären Störungen aufrechtzuerhalten oder den Nutzer angemessen zu informieren. Die Informationen sind sowohl für Entwickler nützlich, die das Backend erweitern oder warten, als auch für das Frontend-Team, um Fehlerszenarien korrekt abzufangen und die Benutzerfreundlichkeit zu gewährleisten. Ziel ist es, Transparenz über die Funktionalität der Endpunkte zu schaffen und die Robustheit des Systems durch sinnvolle Ausfallszenarien und Notfallmaßnahmen zu erhöhen.

### Bundesliga-ChatBot 1.0 OAS 3.1

[openapi.json](#)

Ein spezialisierter ChatBot, der Bundesliga-Daten in Echtzeit über die OpenLigaDB-API bereitstellt. Nutzer können Fragen zu Spielständen, Tabellen, Spielerstatistiken und mehr stellen.

[Kamal Badawi - Website](#)

[Send email to Kamal Badawi](#)

[MIT License](#)

#### default

**GET** / Read Root

**POST** /question Ask Question

**POST** /conversations\_info Post Conversations Info

**DELETE** /delete\_all\_conversations Delete All Conversations

**DELETE** /delete\_conversation\_by\_conversation\_id Delete Conversation By Conversation Id

**POST** /create\_conversation Create Chat Conversation

**POST** /download\_conversation Download Conversation

**POST** /send\_conversation Send Conversation

**POST** /add\_dialog\_item Post Add Dialog Item

**POST** /add\_dialog\_title Post Add Dialog Title

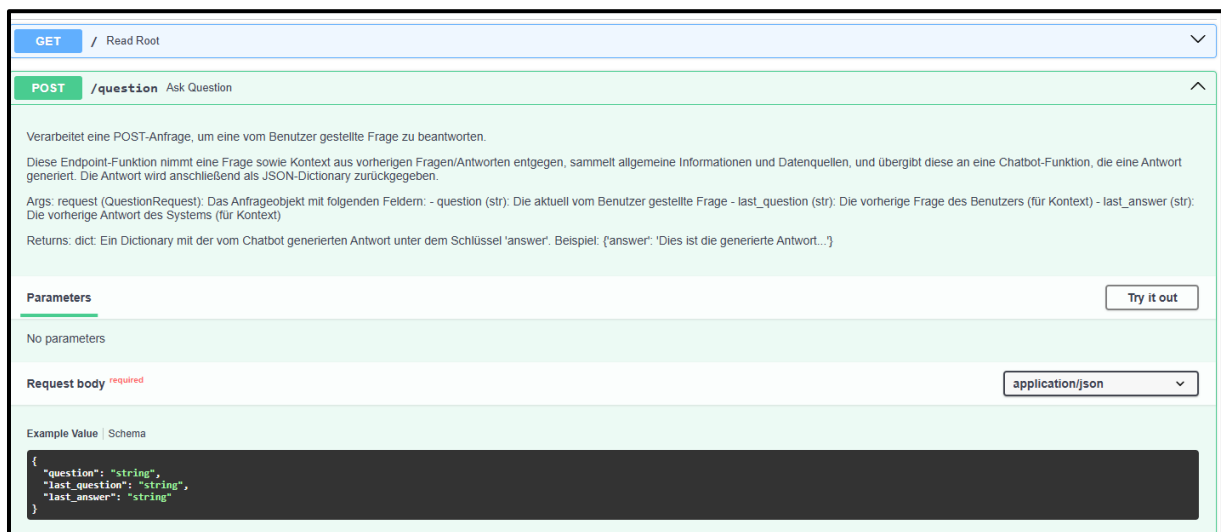
**POST** /text\_to\_speech Post Text To Speech

**POST** /speech\_to\_text Post Speech To Text

**POST** /conversations\_dialogs/{conversation\_id} Post Conversations Dialogs

## /question

- ✚ Dieser Endpunkt dient dazu, eine Nutzerfrage entgegenzunehmen und eine Antwort zur Bundesliga mithilfe von *Google Gemini 2.5 Pro* zu generieren.
- ✚ Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann keine Antwort auf die gestellte Frage geliefert werden.
- ✚ Mögliche Maßnahme: Eine statische Fehlermeldung zurückgeben („Der Dienst ist aktuell nicht verfügbar“), alternativ einen Dummy-Response oder gespeicherte Standard-FAQs anbieten.



The image shows the Swagger UI for the `POST /question` endpoint. It includes a description of the endpoint's purpose, its arguments, and its return type. The 'Parameters' section is empty. The 'Request body' section is marked as 'required' and has a dropdown menu set to 'application/json'. An 'Example Value' is provided in a code block.

**GET** / Read Root

**POST** /question Ask Question

Verarbeitet eine POST-Anfrage, um eine vom Benutzer gestellte Frage zu beantworten.

Diese Endpoint-Funktion nimmt eine Frage sowie Kontext aus vorherigen Fragen/Antworten entgegen, sammelt allgemeine Informationen und Datenquellen, und übergibt diese an eine Chatbot-Funktion, die eine Antwort generiert. Die Antwort wird anschließend als JSON-Dictionary zurückgegeben.

Args: request (QuestionRequest): Das Anfrageobjekt mit folgenden Feldern: - question (str): Die aktuell vom Benutzer gestellte Frage - last\_question (str): Die vorherige Frage des Benutzers (für Kontext) - last\_answer (str): Die vorherige Antwort des Systems (für Kontext)

Returns: dict: Ein Dictionary mit der vom Chatbot generierten Antwort unter dem Schlüssel 'answer'. Beispiel: {'answer': 'Dies ist die generierte Antwort...'}

**Parameters** Try it out


No parameters

**Request body** required application/json

Example Value | Schema

```
{  "question": "string",  "last_question": "string",  "last_answer": "string"}
```

## Request-Beispiel:



The image shows a request example in a tool. It includes a curl command, the request URL, and the server response. The response is a 200 status code with a JSON body and several headers.

**Responses**

**Curl**

```
curl -X 'POST' \  'http://127.0.0.1:8000/question' \  -H 'accept: application/json' \  -H 'Content-Type: application/json' \  -d '{  "question": "Wann war Eintracht Frankfurts letztes Spiel?",  "last_question": " ",  "last_answer": " "  }'
```

**Request URL**

```
http://127.0.0.1:8000/question
```

**Server response**

Code	Details
200	<p><b>Response body</b></p> <pre>{  "answer": "Eintracht Frankfurts letztes Spiel fand am 2025-09-21 statt.\n"</pre> <p><b>Response headers</b></p> <pre>access-control-allow-credentials: true  content-length: 75  content-type: application/json  date: Tue, 23 Sep 2025 18:27:19 GMT  server: uvicorn</pre>

Nächste Frage stellen:

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/question' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "question": "Gegen wen hat Eintracht Frankfurt gespielt?",
    "last_question": "Mann war Eintracht Frankfurts letztes Spiel?",
    "last_answer": "Eintracht Frankfurts letztes Spiel fand am 2025-09-21 statt.\n"
  }'
```

Request URL

http://127.0.0.1:8000/question

Server response

Code	Details
200	<div><div>Response body</div><pre>{   "answer": "Eintracht Frankfurt spielte gegen 1. FC Union Berlin.\n" }</pre><div> <a href="#">Download</a></div></div> <div><div>Response headers</div><pre>access-control-allow-credentials: true content-length: 68 content-type: application/json date: Tue, 23 Sep 2025 18:46:20 GMT server: uvicorn</pre></div>

## /create\_conversation

- 🔧 Dieser Endpunkt erstellt eine neue Konversation für einen bestimmten Nutzer anhand der `user_id`. Dies geschieht automatisch, wenn der Nutzer den Bundesliga-ChatBot öffnet oder den Button „Neuer Chat“ betätigt.
- 🔧 Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann keine neue Unterhaltung gestartet werden.
- 🔧 Mögliche Maßnahme: Fehlermeldung an Nutzer; alternativ Konversation zunächst im Frontend puffern und bei Wiederverfügbarkeit in die Datenbank schreiben.

POST /create\_conversation Create Chat Conversation

Erstellt eine neue, leere Konversation für einen Benutzer.

Parameter:

- request (ConversationInputRequest): Objekt mit der `user_id`.

Ablauf:

- Generiert eine eindeutige `conversation_id`.
- Initialisiert eine leere Konversation ohne Titel und ohne Dialogeinträge.
- Speichert die Konversation mit `create_conversation`.
- Gibt eine HTTP 409-Fehlermeldung zurück, wenn bereits eine Konversation mit der gleichen ID existiert.

Rückgabewert:

- JSON-Antwort mit Bestätigungsnachricht und generierter `conversation_id`.

Parameters

No parameters

Request body required

application/json

Example Value | Schema

```
{
  "user_id": "string"
}
```

## Request-Beispiel:

### Responses

#### Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/create_conversation' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "user_id": "user_001"
  }'
```

#### Request URL

```
http://127.0.0.1:8000/create_conversation
```

#### Server response

Code	Details
200	<div><h5>Response body</h5><pre>{   "message": "Conversation added successfully.",   "conversation_id": "mnjEft4RIaPqPx02050923202907813UYav8c2hHwKu9r4" }</pre><div> <a href="#">Download</a></div></div> <div><h5>Response headers</h5><pre>access-control-allow-credentials: true content-length: 114 content-type: application/json date: Tue, 23 Sep 2025 18:29:07 GMT server: uvicorn</pre></div>

## /add\_dialog\_title

- 🚀 Dieser Endpunkt erstellt anhand des ersten Nachrichtenpaars (Frage und Antwort) zu einer bestimmten Konversation (`conversation_id`) und einem Nutzer (`user_id`) mithilfe von *Google Gemini 2.5 Pro* einen passenden Titel, der in der Chat-Historie angezeigt wird. Dies geschieht automatisch, wenn der Nutzer seine erste Frage abschickt und die Antwort vom Backend (*Google Gemini 2.5 Pro*) erhält.
- 🚀 Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann kein Titel für die Konversation erstellt werden.
- 🚀 Mögliche Maßnahme: Generischen Standardtitel vergeben („Neue Unterhaltung“).

**POST** /add\_dialog\_title Post Add Dialog Title

Generiert und speichert einen Titel basierend auf Fragen und Antworten für eine Konversation.

Parameter:

- request (DialogTitleInputRequest): Benutzer-ID, Konversations-ID und QA-Paare.

Rückgabewert:

- JSON-Antwort mit dem generierten Titel oder Fehler, wenn Konversation nicht existiert.

**Parameters** [Try it out](#)

No parameters

**Request body** required application/json

**Example Value** | **Schema**

```
{
  "user_id": "string",
  "conversation_id": "string",
  "questions_and_answers": [
    {
      "question": "string",
      "answer": "string"
    }
  ]
}
```



## Request-Beispiel:

### Responses


#### Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/add_dialog_title' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "user_id": "user_001",
    "conversation_id": "mnjEFt4RIaPqPx020250923202907813UYavBc2hHwKu9r4",
    "questions_and_answers": [
      {
        "question": "Wann war Eintracht Frankfurts letztes Spiel?",
        "answer": "Eintracht Frankfurts letztes Spiel fand am 2025-09-21 statt.\n"
      }
    ]
  }'
```

#### Request URL

```
http://127.0.0.1:8000/add_dialog_title
```

#### Server response

Code	Details
200	<div><p>Response body</p><pre>{   "message": "Eintracht Frankfurt Spiel" }</pre><div> <a href="#">Download</a></div><p>Response headers</p><pre>access-control-allow-credentials: true content-length: 39 content-type: application/json date: Tue, 23 Sep 2025 18:37:33 GMT server: uvicorn</pre></div>

## /conversations\_info

- ✚ Dieser Endpunkt gibt alle Konversationstitel, Datum, Uhrzeit (Chat-Historie) für einen bestimmten Nutzer anhand der `user_id` zurück.
- ✚ Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann der Nutzer seine Chat-Historie nicht einsehen.
- ✚ Mögliche Maßnahme: Hinweis an den Nutzer, dass die Historie gerade nicht verfügbar ist; Zwischenlösung wäre das Zwischenspeichern im Frontend (LocalStorage).

### POST /conversations\_info Post Conversations Info

Lädt die Gesprächsinformationen eines Benutzers (Titel, ID, Datum, Uhrzeit).

Parameter:

- request (ConversationInfoRequest): Objekt mit der `user_id`.

Ablauf:

- Ruft die Funktion `get_conversations_info` mit der `user_id` auf.
- Gibt eine 404-Fehlermeldung zurück, wenn keine Daten gefunden werden.
- Gibt bei Erfolg eine Liste mit Gesprächsinformationen als JSON zurück.

Rückgabewert:

- Liste von Objekten mit 'title', 'conversation\_id', 'date', 'time'.

#### Parameters

No parameters

#### Request body required

application/json

#### Example Value | Schema

```
{
  "user_id": "string"
}
```

## Request-Beispiel:

### Responses


#### Curl

```
curl -X 'POST' \
'http://127.0.0.1:8000/conversations_info' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "user_id": "user_001"
}'
```

#### Request URL

```
http://127.0.0.1:8000/conversations_info
```

#### Server response

Code	Details
200	<div><h5>Response body</h5><pre>{   "conversations": [     {       "title": "Eintracht Frankfurt Spiel",       "conversation_id": "mnjEFt4RIaPqPx020250923202907813UYavBc2hHWku9r4",       "date": "23-09-2025",       "time": "20:29:07"     }   ] }</pre><div> <a href="#">Download</a></div></div> <div><h5>Response headers</h5><pre>access-control-allow-credentials: true content-length: 163 content-type: application/json date: Tue, 23 Sep 2025 18:41:36 GMT server: uvicorn</pre></div>

## /send\_conversation

- ✚ Dieser Endpunkt sendet eine bestehende Konversation anhand der `conversation_id` und der vom Nutzer hinterlegten E-Mail-Adresse als `PDF`-Datei per E-Mail. Dies geschieht automatisch, wenn der Nutzer den Send-Button für eine bestimmte Konversation in der Chat-Historie betätigt.
- ✚ Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann der Nutzer die Konversation nicht als `PDF`-Datei per E-Mail an sich selbst senden.
- ✚ Mögliche Maßnahme: Fehlermeldung mit Hinweis, stattdessen den PDF-Download zu nutzen; späteres automatisches Nachsenden ermöglichen.

**POST** /send\_conversation Send Conversation

Sendet ein Gespräch per E-Mail als PDF-Anhang.

Parameter:

- request (SendConversationRequest): Objekt mit `conversation_id` und `email_address`.

Ablauf:

- Holt die Konversation.
- Erstellt ein PDF.
- Sendet das PDF an die angegebene E-Mail-Adresse.

Rückgabewert:

- JSON-Antwort mit Bestätigung.

Parameters

No parameters

Try it out

Request body required

application/json

Example Value | Schema

```
{
  "conversation_id": "string",
  "email_address": "string"
}
```

## Request-Beispiel:

### Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/send_conversation' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "conversation_id": "mnjEFt4RIaPqPx020250923202907813UVavBc2hHwKu9r4",
    "email_address": "abdal.ahmad@outlook.de"
  }'
```

Request URL

http://127.0.0.1:8000/send\_conversation

Server response

Code	Details
200	<p>Response body</p> <pre>{   "message": "Report sent to abdal.ahmad@outlook.de successfully" }</pre> <p>Response headers</p> <pre>access-control-allow-credentials: true content-length: 64 content-type: application/json date: Tue, 23 Sep 2025 18:53:56 GMT server: uvicorn</pre>

## /download\_conversation

- 🚦 Dieser Endpunkt lädt eine bestehende Konversation anhand der `conversation_id` als PDF-Datei lokal herunter. Dies erfolgt automatisch, wenn der Nutzer den Download-Button für eine bestimmte Konversation in der Chat-Historie betätigt.
- 🚦 Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann der Nutzer die Konversation nicht als PDF-Datei herunterladen.
- 🚦 Mögliche Maßnahme: Nutzerhinweis; optional Export im Frontend (z. B. HTML → PDF) anbieten.

### POST /download\_conversation Download Conversation

Lädt ein Gespräch im PDF-Format herunter.

Parameter:

- request (DownloadConversationRequest): Objekt mit conversation\_id.

Ablauf:

- Holt die Konversationsdaten über `get_conversations_dialogs`.
- Erzeugt ein PDF über `Create PDF`.
- Gibt die PDF-Datei als Streaming-Antwort zurück.

Rückgabewert:

- StreamingResponse: PDF-Download als Stream.

Parameters

No parameters

Request body required

application/json

Example Value | Schema

```
{
  "conversation_id": "string"
}
```

## Request-Beispiel:

### Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/download_conversation' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "conversation_id": "mnjEFt4RIaPqPx020250923202907813UVavBc2hHMKu9r4"
  }'
```

Request URL

```
http://127.0.0.1:8000/download_conversation
```

Server response

Code	Details
200	<p>Response body</p> <p><a href="#">Download file</a></p> <p>Response headers</p> <pre>access-control-allow-credentials: true content-disposition: attachment; filename="23-09-2025-20-57-38.pdf" content-type: application/pdf date: Tue, 23 Sep 2025 18:57:37 GMT server: uvicorn transfer-encoding: chunked</pre>

## /add\_dialog\_item

- ✚ Dieser Endpunkt fügt ein Nachrichtenpaar (Frage und Antwort) zu einer bestimmten Konversation (`conversation_id`) und einem Nutzer (`user_id`) hinzu. Dies geschieht automatisch, wenn der Nutzer seine Frage abschickt und die Antwort vom Backend (*Google Gemini 2.5 Pro*) erhält.
- ✚ Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann kein Frage-Antwort-Paar zu einer bestehenden oder neu erstellten Konversation hinzugefügt werden.
- ✚ Mögliche Maßnahme: Temporär im Frontend speichern und später nachtragen; Nutzerhinweis „Antwort wurde nicht gespeichert“.

### POST /add\_dialog\_item Post Add Dialog Item

Fügt einen einzelnen Frage-Antwort-Dialogeintrag zu einer bestehenden Konversation hinzu.

Parameter:

- request (DialogItemInputRequest): Benutzer-ID, Konversations-ID, Frage, Antwort, Datum und Uhrzeit.

Rückgabewert:

- JSON-Antwort bei Erfolg, 404-Fehler wenn Konversation nicht existiert.

Parameters

No parameters

Request body required

application/json

Example Value | Schema

```
{
  "user_id": "string",
  "conversation_id": "string",
  "question": "string",
  "answer": "string",
  "date": "string",
  "time": "string"
}
```

## Request-Beispiel:

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/add_dialog_item' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "user_id": "user_001",
    "conversation_id": "mnjEft4RIaPqPx020250923202907813UYavBc2hHWKu9r4",
    "question": "Wann war Eintracht Frankfurts letztes Spiel?",
    "answer": "Eintracht Frankfurts letztes Spiel fand am 2025-09-21 statt.\n",
    "date": "23.09.2025",
    "time": "21:02"
  }'
```

Request URL

http://127.0.0.1:8000/add\_dialog\_item

Server response

Code	Details
200	<p>Response body</p> <pre>{   "message": "Dialog item added successfully." }</pre> <p>Response headers</p> <pre>access-control-allow-credentials: true content-length: 45 content-type: application/json date: Tue, 23 Sep 2025 19:03:09 GMT server: uvicorn</pre>

## /conversations\_dialogs/{conversation\_id}

- 🚩 Dieser Endpunkt liefert den Konversationsdialog zu einer bestimmten `conversation_id` (Chat-Dialog) zurück.
- 🚩 Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann der Nutzer die Fragen und Antworten von *Google Gemini 2.5 Pro* für die betreffende Konversation nicht einsehen.
- 🚩 Mögliche Maßnahme: Nutzerhinweis anzeigen; bei lokalem Cache die letzte geladene Version anbieten.

**POST** /conversations\_dialogs/{conversation\_id} Post Conversations Dialogs

Gibt die Dialogeinträge einer Konversation formatiert für das Frontend zurück.

Parameter:

- conversation\_id (str): Die ID der Konversation.

Rückgabewert:

- JSON-Antwort mit formatierten Dialogeinträgen oder 404-Fehlermeldung.

Parameters

Name	Description
conversation_id * required string (path)	conversation_id

Try it out

Request-Beispiel:

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/conversations_dialogs/mnjEft4RIaPqPx020250923202907813UYavBc2hHwKu9r4' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

http://127.0.0.1:8000/conversations\_dialogs/mnjEft4RIaPqPx020250923202907813UYavBc2hHwKu9r4

Server response

Code	Details
200	<p>Response body</p> <pre>{   "conversations_dialogs": [     {       "question": "Wann war Eintracht Frankfurts letztes Spiel?",       "answer": "Eintracht Frankfurts letztes Spiel fand am 2025-09-21 statt.\n",       "date": "23.09.2025",       "time": "21:02"     }   ] }</pre> <p>Response headers</p> <pre>access-control-allow-credentials: true content-length: 196 content-type: application/json date: Tue, 23 Sep 2025 19:07:43 GMT server: uvicorn</pre>

## /text\_to\_speech

- ✚ Dieser Endpunkt konvertiert die Antwort von *Google Gemini 2.5 Pro* (in Textform) mithilfe der *VoiceRSS-API* in ein Audioformat und gibt die Audiodatei an das Frontend zurück.
- ✚ Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann keine Audiodatei zurückgegeben werden.
- ✚ Mögliche Maßnahme: Textausgabe anbieten („Audioausgabe derzeit nicht verfügbar“).

## Request-Beispiel:

**POST** /text\_to\_speech Post Text To Speech

Wandelt Text in gesprochene Sprache um und gibt ihn als Stream zurück.

Args: text (str): Der zu sprechende Text.

Returns: StreamingResponse: Gestreamte Audiodatei.

Parameters

Name	Description
text * required string (query)	<input type="text" value="text"/>

Try it out

Text: „Wann spielt Eintracht gegen Bayern?“

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/text_to_speech?text=Wann%20spielt%20Eintracht%20gegen%20Bayern%3F' \
  -H 'accept: application/json' \
  -d ''
```

Request URL

```
http://127.0.0.1:8000/text_to_speech?text=Wann%20spielt%20Eintracht%20gegen%20Bayern%3F
```

Server response

Code	Details
200	<p>Response body</p> <p><a href="#">Download file</a></p> <p>Response headers</p> <pre>access-control-allow-credentials: true content-type: application/octet-stream date: Fri, 26 Sep 2025 20:27:39 GMT server: uvicorn transfer-encoding: chunked</pre>

Responses

Code	Description	Links
200	Successful Response	No links

Media type

application/json

Controls: Accept header

## /speech\_to\_text

- ✚ Dieser Endpunkt konvertiert die vom Nutzer gestellte Frage (in Audioform) mithilfe der *AssemblyAI-API* in Textform, da *Google Gemini 2.5 Pro* im Bundesliga-ChatBot nur mit Texten arbeitet.
- ✚ Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann keine Audiodatei in Text konvertiert werden.
- ✚ Mögliche Maßnahme: Nutzer auffordern, die Frage manuell einzutippen.

Request-Beispiel:

Audiodatei hochladen

Parameters

No parameters

Request body **required**

multipart/form-data

speech \* **required**

string(\$binary)

Datei auswählen audio.mp3

Execute

Clear

Cancel

Reset

Responses

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:8000/speech_to_text' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'speech=@audio.mp3;type=audio/mpeg'
```

Request URL

http://127.0.0.1:8000/speech\_to\_text

Server response

Code	Details
200	<p>Response body</p> <pre>{   "text": "Wann spielt Bayern München gegen Eintracht Frankfurt?" }</pre> <p>Response headers</p> <pre>access-control-allow-credentials: true content-length: 65 content-type: application/json date: Fri, 26 Sep 2025 20:40:32 GMT server: uvicorn</pre>

Responses

## /delete\_conversation\_by\_conversation\_id

- ✚ Dieser Endpunkt löscht eine bestimmte Konversation anhand der `conversation_id` und entfernt somit den gesamten Dialog dieser Unterhaltung.
- ✚ Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann der Nutzer die jeweilige Konversation nicht löschen.
- ✚ Mögliche Maßnahme: Temporär nur als „archiviert“ markieren im Frontend und später synchronisieren.

**DELETE** /delete\_conversation\_by\_conversation\_id Delete Conversation By Conversation Id

Löscht eine spezifische Konversation basierend auf der `conversation_id`.

Parameter:

- request (DeleteConversationByConversationIdRequest): Objekt mit dem Feld `conversation_id`.

Ablauf:

- Ruft die Funktion `remove_conversation_by_conversation_id` auf, um die Konversation zu löschen.
- Gibt eine HTTP 404-Fehlermeldung zurück, wenn keine passende Konversation gefunden oder gelöscht wurde.

Rückgabewert:

- JSON-Antwort mit einer Bestätigungsmeldung bei erfolgreichem Löschen.

Parameters

No parameters

Request body required

application/json

Example Value | Schema

```
{
  "conversation_id": "string"
}
```

Request-Beispiel:



**Responses**

**Curl**

```
curl -X 'DELETE' \
  'http://127.0.0.1:8000/delete_conversation_by_conversation_id' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "conversation_id": "mnjEft4RIaPqPx20250923202907813UYavBc2hMkKu9r4"
  }'
```

**Request URL**

```
http://127.0.0.1:8000/delete_conversation_by_conversation_id
```

**Server response**

Code	Details
200	<p><b>Response body</b></p> <pre>{   "message": "Conversation 'mnjEft4RIaPqPx20250923202907813UYavBc2hMkKu9r4' has been removed" }</pre> <p><b>Response headers</b></p> <pre>access-control-allow-credentials: true content-length: 93 content-type: application/json date: Fri, 26 Sep 2025 20:53:10 GMT server: uvicorn</pre>

**Responses**

## /delete\_all\_conversations

- ✚ Dieser Endpunkt löscht alle Konversationen eines bestimmten Nutzers anhand der `user_id` und leert somit die Chat-Historie dieses Nutzers.
- ✚ Ausfallprognose: Sollte dieser Endpunkt ausfallen, kann der Nutzer seine Chat-Historie nicht löschen.
- ✚ Mögliche Maßnahme: Nutzer informieren und Löschung zu einem späteren Zeitpunkt erneut versuchen (z. B. Warteschlange im Backend).

**DELETE** /delete\_all\_conversations Delete All Conversations

Löscht alle Konversationen eines bestimmten Benutzers.

**Parameter:**

- request (DeleteAllConversationsRequest): Objekt mit dem Feld `user_id`.

**Ablauf:**

- Ruft die Funktion `remove_all_conversations` auf, um alle Gespräche des angegebenen Benutzers zu löschen.
- Gibt eine HTTP 404-Fehlermeldung zurück, wenn der Benutzer nicht existiert oder keine Konversationen zum Löschen vorhanden sind.
- Gibt eine Bestätigung auf der Konsole aus und sendet eine Erfolgsnachricht zurück.

**Rückgabewert:**

- JSON-Antwort mit einer Bestätigung, dass alle Konversationen des Benutzers gelöscht wurden.

**Parameters** Try it out

No parameters

**Request body** required application/json

**Example Value** | **Schema**

```
{
  "user_id": "string"
}
```

Request-Beispiel:



## 6. API-Keys erstellen

### 6.1. Gmail-Passkey erstellen

Damit eine bestimmte Konversation vom Bundesliga-ChatBot als PDF-Datei per *Python* und *Gmail* verschickt werden kann, wird ein Passkey benötigt, der zusammen mit der E-Mail-Adresse in der *.env*-Datei gespeichert wird.

**Sicherheit: 2-Faktor-Authentifizierung (2FA) muss aktiviert werden:**

- 🚩 Das Google-Konto wird geöffnet und der Bereich [Sicherheit](#) aufgerufen.
- 🚩 Dort wird die 2-Faktor-Authentifizierung aktiviert.
- 🚩 Diese Sicherheitsmaßnahme wird als Voraussetzung für die anschließende Erstellung eines App-Passkeys benötigt.

**Passkey (App-Passwort) wird erstellt:**

- 🚩 Nach der Aktivierung der 2FA wird im Bereich App-Passwörter die Option gefunden, ein neues Passwort zu generieren.
- 🚩 Als App wird „Mail“ und als Gerät „Anderes“ (z. B. „*Python*-Skript“) ausgewählt.
- 🚩 Ein spezieller 16-stelliger Passkey wird von Google generiert.

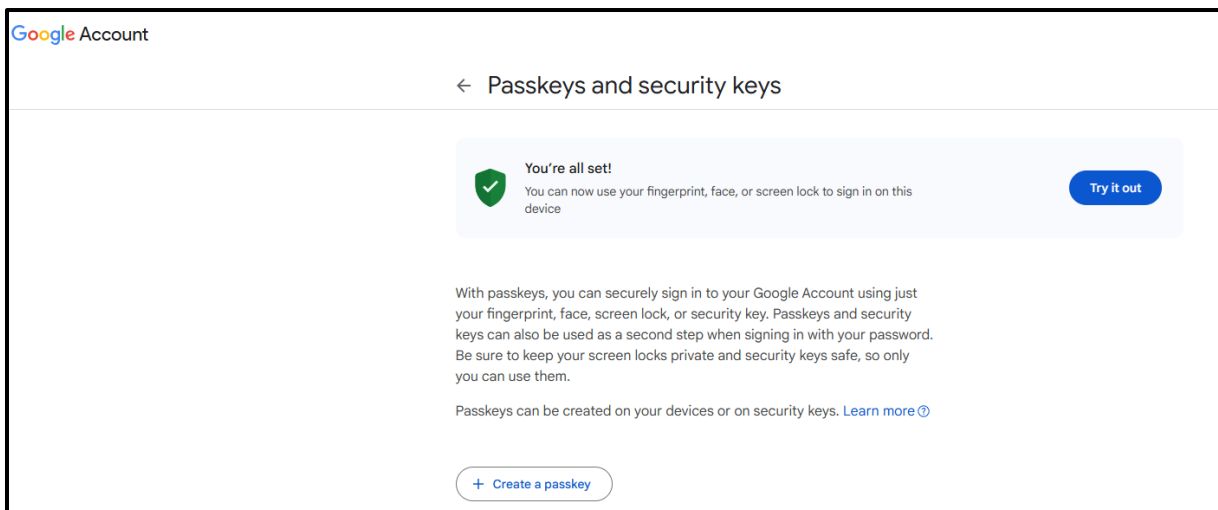
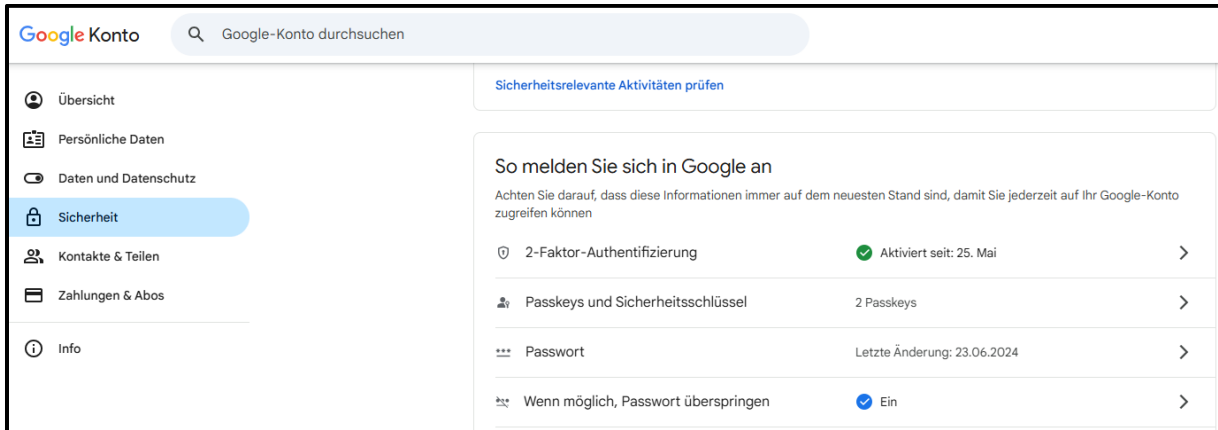
**Wichtig:** Das normale Google-Passwort darf nicht verwendet werden, sondern ausschließlich dieser Passkey für den Mailversand aus *Python*.

**Passkey & E-Mail werden in der *.env*-Datei gespeichert:**

- Die *Gmail*-Adresse und der generierte Passkey müssen in der `.env`-Datei gespeichert werden, damit darauf vom *Python*-Skript zugegriffen werden kann.

```
FROM_EMAIL=email_address
```

```
FROM_EMAIL_PASSWORD=gmail_pass_key
```




## 6.2. Kostenlosen *Google Gemini 2.5 Pro* API-Key erstellen

*Google Gemini 2.5 Pro* gilt als der Motor für den Bundesliga-ChatBot, mit dem die Fragen von Nutzern anhand der verfügbaren Daten beantwortet werden. Von Google wird zum aktuellen Zeitpunkt (mit Vorbehalt) die Möglichkeit angeboten, einen kostenlosen API-Key für *Google Gemini 2.5 Pro* zu erhalten.

Die Erstellung eines kostenlosen API-Keys für *Google Gemini 2.5 Pro* erfolgt wie folgt:

- Unter der Webseite wird der folgende Link aufgerufen: <https://ai.google.dev/gemini-api/docs/api-key?hl=de>
- Im Google AI Studio wird die Option „API-Schlüssel erstellen“ angeklickt.
- Der erzeugte Schlüssel wird kopiert und in der `.env`-Datei hinzugefügt.

```
GOOGLE_GEMINI_API_KEY=api_key
```

Startseite > Gemini API > Modelle
War das hilfreich?  

## Gemini API-Schlüssel verwenden


Feedback geben

Zur Verwendung der Gemini API benötigen Sie einen API-Schlüssel. Sie können in [Google AI Studio](#) mit wenigen Klicks einen kostenlosen Schlüssel erstellen.

API-Schlüssel
+ API-Schlüssel erstellen




Gemini API schnell testen

[API-Kurzanleitung](#)



Use code with caution.

Ihre API-Schlüssel sind unten aufgeführt. Sie können Ihr Projekt und Ihre API-Schlüssel auch in Google Cloud ansehen und verwalten.

Projektnummer	Projektname	API-Schlüssel	Erstellt	Tarif
...7154	Gemini API 		04.02.2025	Free <a href="#">Abrechnung einrichten</a> <a href="#">Nutzungsdaten ansehen</a> 

### 6.3. VoiceRSS-API Key erstellen

Voice RSS bietet die Möglichkeit, Text in Audios zu konvertieren. Diese Funktion kann genutzt werden, um die Antworten aus *Google Gemini 2.5 Pro* für den Bundesliga-ChatBot von Text in Audio umzuwandeln. Von *VoiceRSS* wird zum aktuellen Zeitpunkt (mit Vorbehalt) die Möglichkeit angeboten, einen kostenlosen *VoiceRSS-API-Key* für die *VoiceRSS -API* zu erhalten.

Die Vorgehensweise ist wie folgt:

- Die Webseite <https://www.voicerss.org/> wird aufgerufen.
- Eine Registrierung wird durchgeführt.
- Der API-Key wird kopiert und in der `.env`-Datei hinzugefügt.

```
VOICERSS_API_KEY=api_key
```

# Voice RSS

**FREE**

**Monthly Subscription** \$0.00

#Daily Request 350  
(Limited to 100KB per request)

Plain text input YES

SSML input NO

Basic Dashboard (Statistics) NO

Premium Dashboard NO

SLA of 0.5 Sec response NO

**GET STARTED**




## Registration

For registration please fill the following form.

First name	Last name
<input type="text"/>	<input type="text"/>
Company name	E-mail *
<input type="text"/>	<input type="text"/>
Password *	Confirm password *
<input type="text"/>	<input type="text"/>
Site URL	
<input type="text"/>	

\* - mandatory fields

☐ Ich bin kein Roboter. 

reCAPTCHA  
Datenschutzrichtlinie · Nutzungsbedingungen

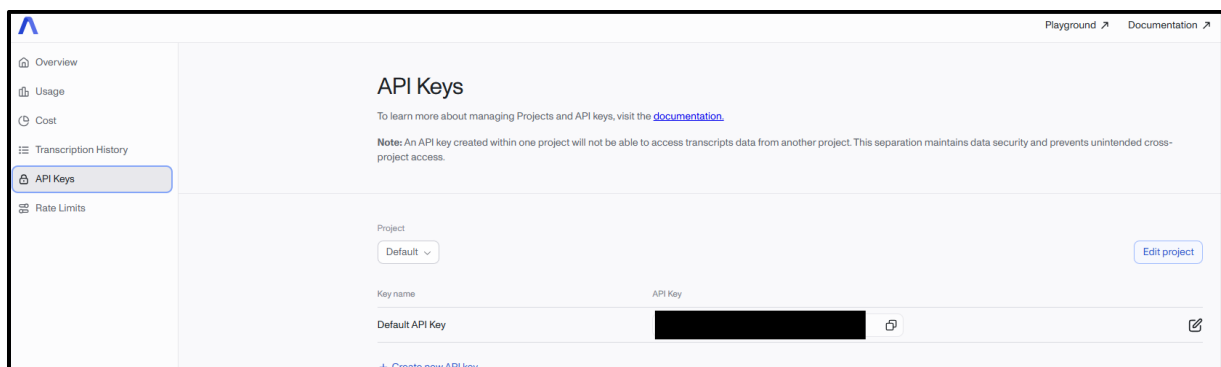
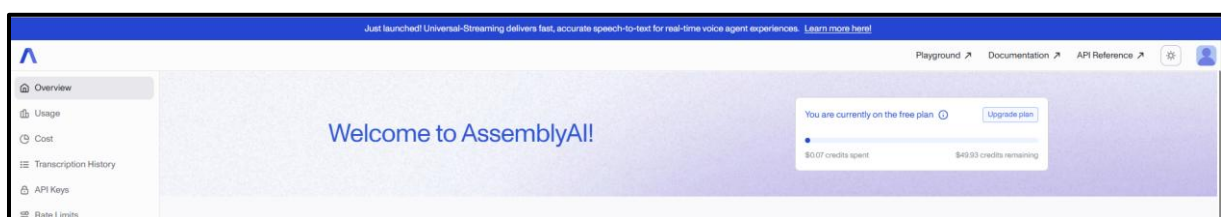
## 6.4 AssemblyAI-API Key erstellen

AssemblyAI bietet die Möglichkeit, Audios in Texte zu konvertieren. Diese Funktion kann genutzt werden, um gesprochene Fragen von Nutzern in Textform umzuwandeln und anschließend an *Google Gemini 2.5 Pro* zur Beantwortung im Bundesliga-ChatBot weiterzugeben. Von AssemblyAI wird zum aktuellen Zeitpunkt (mit Vorbehalt) die Möglichkeit angeboten, einen kostenlosen API-Key zu erhalten.

Die Vorgehensweise ist wie folgt:

- Die Webseite <https://www.assemblyai.com/dashboard/overview> wird aufgerufen.
- Eine Registrierung wird durchgeführt.
- Im Bereich „API Keys“ kann ein kostenloser Schlüssel erstellt werden.
- Der generierte API-Key wird kopiert und in der `.env`-Datei gespeichert.

```
ASSEMBLYAI_API_KEY=api_key
```



## 6.5 MongoDB Key erstellen

*MongoDB* bietet eine flexible Möglichkeit zur Speicherung und Verwaltung von Daten, z. B. für Chatverläufe oder Nutzeranfragen im Bundesliga-ChatBot.

Für die Entwicklung kann ein kostenloser Zugang zu *MongoDB* Atlas genutzt werden - dem Cloud-Dienst von *MongoDB*. Die Verbindung erfolgt dabei über Umgebungsvariablen, die sicher in einer `.env`-Datei gespeichert werden.

Die Vorgehensweise ist wie folgt:

- Die Webseite <https://www.mongodb.com/cloud/atlas> wird aufgerufen.
- Eine Registrierung wird durchgeführt.
- Ein neues Projekt sowie eine Datenbank-Instanz („Cluster“) wird erstellt.
- Unter „Database Access“ wird ein Benutzer mit Passwort angelegt (z. B. `mongo_db_username` und `mongo_db_password`).
- Unter „Network Access“ wird die eigene IP-Adresse freigegeben (z. B. `0.0.0.0/0` für alle Adressen während der Entwicklung).
- Der Verbindungs-String wird unter „Connect > Drivers > Node.js“ generiert.
- In der `.env`-Datei werden die Zugangsdaten wie folgt gespeichert:

```
MONGO_DB_USERNAME=mongo_db_username
```

```
MONGO_DB_PASSWORD=mongo_db_password
```

### Upgrade Cluster0

Choose a template below to upgrade your cluster or edit additional configuration.  
Cluster configuration can be edited at any time.

☐ **M10** \$0.09/hour

RECOMMENDED

Dedicated Clusters for development environments and low-traffic applications.

STORAGE	RAM	vCPU
10 GB	2 GB	2 vCPUs

**Features**

- ✓ Flexible backups ⓘ
- ✓ Zero-downtime cluster scaling ⓘ
- ✓ Performance diagnostic tools ⓘ
- ✓ Additional search indexes
- ✓ Uptime SLA ⓘ

☐ **Flex** From \$0.011/hour  
Up to \$30/month

MINIMUM UPGRADE

For application development and testing, with on-demand burst capacity for unpredictable traffic.

STORAGE	RAM	vCPU
5 GB	Shared	Shared

**Features**

- ✓ Basic backups ⓘ
- ✓ Upgradeable ⓘ
- ✓ Pay As You Go pricing
- ✓ On-demand burst capacity

☒ **Free**

CURRENT

For learning and exploring MongoDB in a cloud environment.

STORAGE	RAM	vCPU
512 MB	Shared	Shared

**Features**

- ✓ Free forever

## 6.6 OpenLigaDB-API

Die *OpenLigaDB* stellt eine Schnittstelle für Spielinformationen zur Verfügung, z. B. für Bundesliga-Spiele. Da für den Zugriff kein API-Key notwendig ist, genügt es, die Basis-URL zentral in der `.env`-Datei zu speichern.

Dies hat folgende Vorteile:

- keine Hardcodierung im Code,
- einfache Wartung bei Änderungen der URL,
- einheitliche Konfiguration für verschiedene Umgebungen (z. B. Entwicklung, Test, Produktion).

Die Vorgehensweise ist wie folgt:

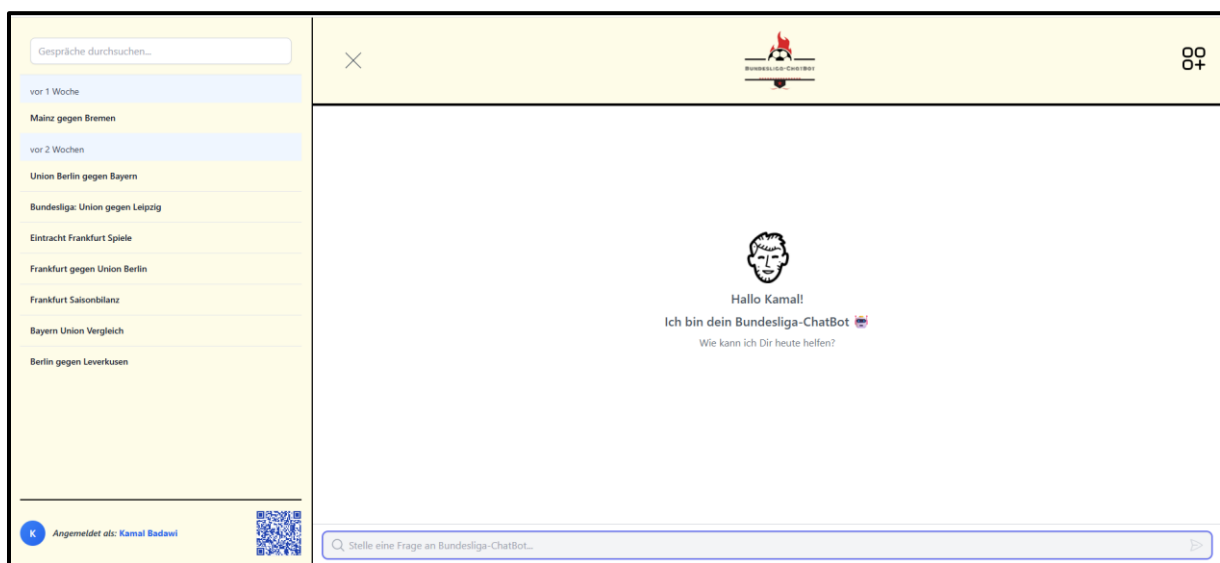
- Die Webseite <https://api.openligadb.de> wird als Datenquelle genutzt.
- Ein API-Key oder eine Registrierung ist nicht erforderlich.
- In der `.env`-Datei wird die Basis-URL wie folgt hinterlegt:

```
DATA_SOURCE=api.openligadb.de
```

Im Backend wird diese Umgebungsvariable eingelesen und für alle API-Anfragen verwendet.

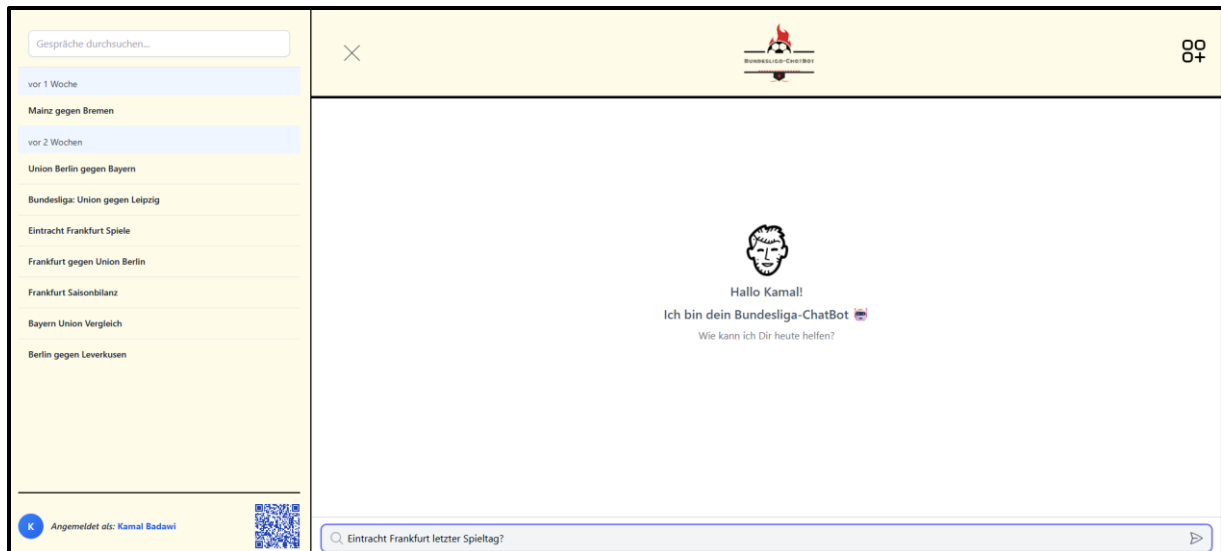
## 7. Anwendungsbeispiel

Beim ersten Aufruf ist der Bundesliga-Chatbot ohne Inhalte im Frage-und-Antwort-Bereich.

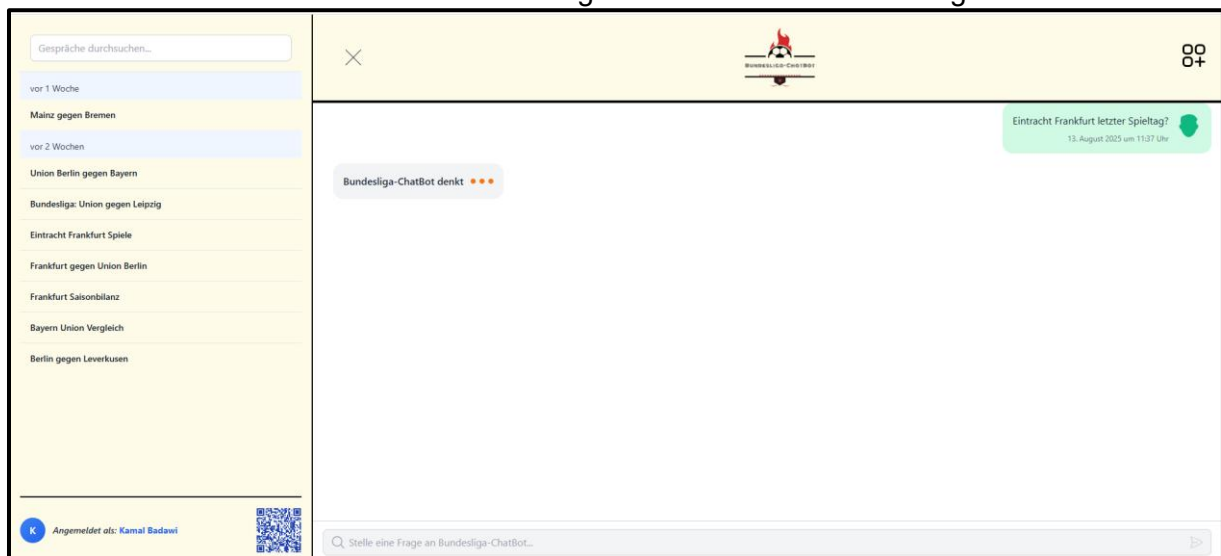


Unten auf der Seite lässt sich eine Frage an den Bundesliga-Chatbot stellen - die Eingabe wird durch Drücken der Enter-Taste oder Klicken auf den Sende-Button bestätigt.

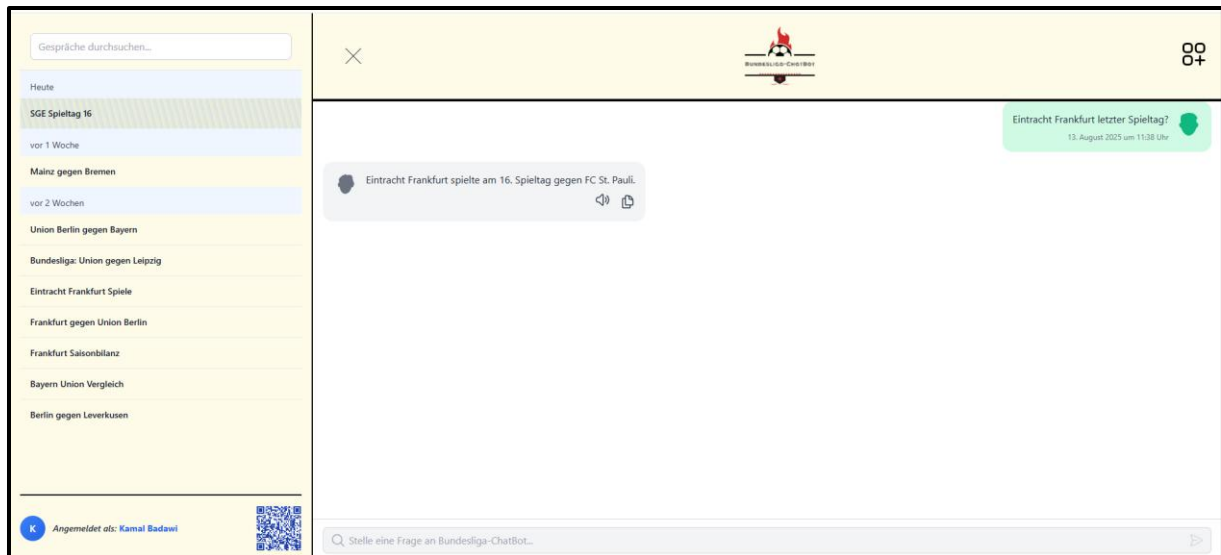




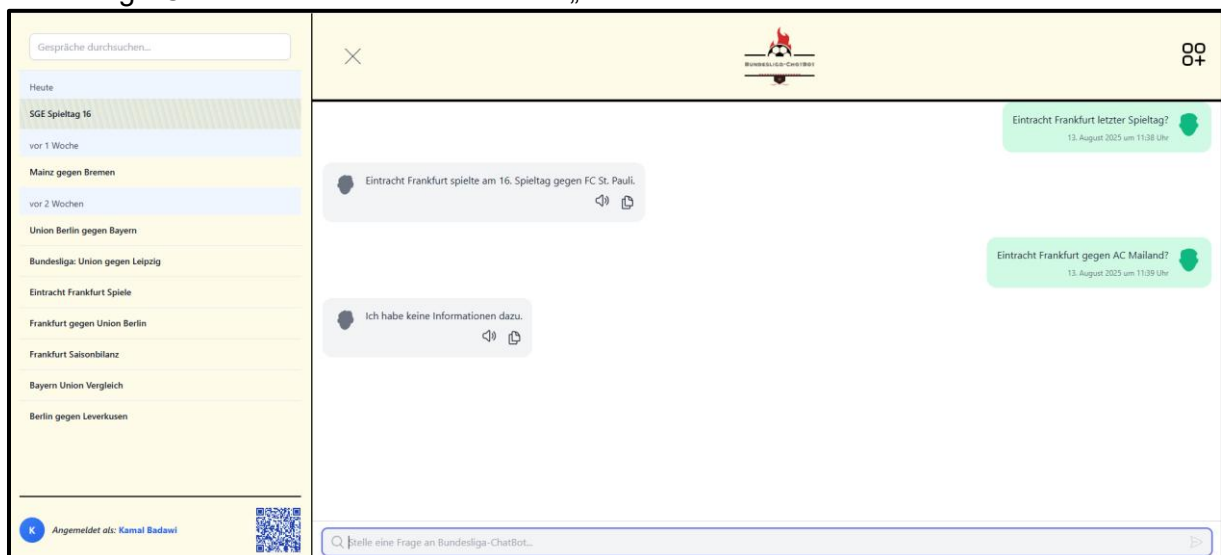
Sobald eine Frage gesendet wurde, wird sie an *Google Gemini 2.5 Pro* weitergeleitet. In der Zwischenzeit erscheint eine Animation mit dem Text „Bundesliga-ChatBot denkt ...“, die nach Abschluss der Antwortverarbeitung automatisch ausgeblendet wird.



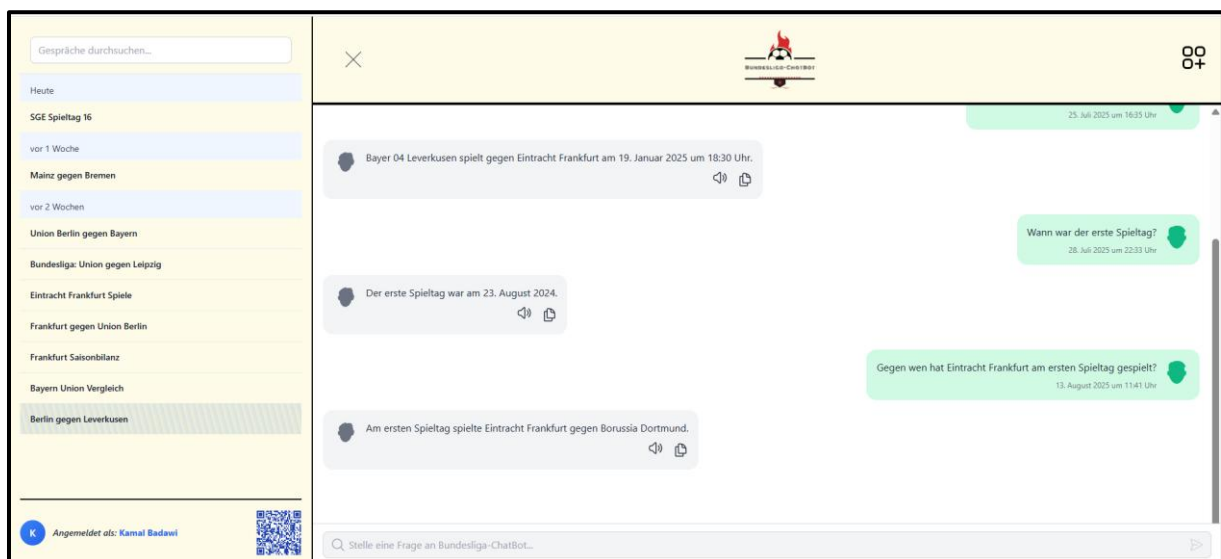
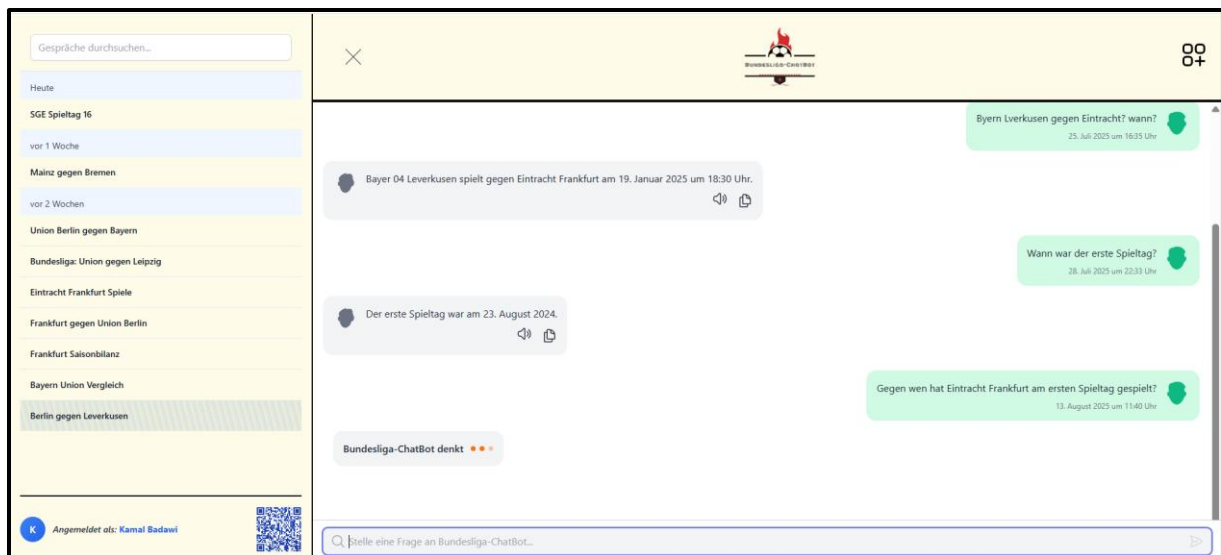
Sobald die Antwort vorliegt, wird gemeinsam mit der ursprünglichen Frage eine Methode aufgerufen, die mithilfe von *Google Gemini 2.5 Pro* automatisch einen Titel für den Chat generiert. Außerdem wird dem Chat eine `conversation_id` zugewiesen. Anschließend werden der Chat - bestehend aus `user_id`, Titel, allen Nachrichten sowie Datum und Uhrzeit - in der *MongoDB* gespeichert, wobei der Chatverlauf absteigend nach Datum und Uhrzeit sortiert wird. Der Nutzer hat jederzeit die Möglichkeit, über einen Button im oberen rechten Bereich des Interfaces einen neuen Chat zu starten.



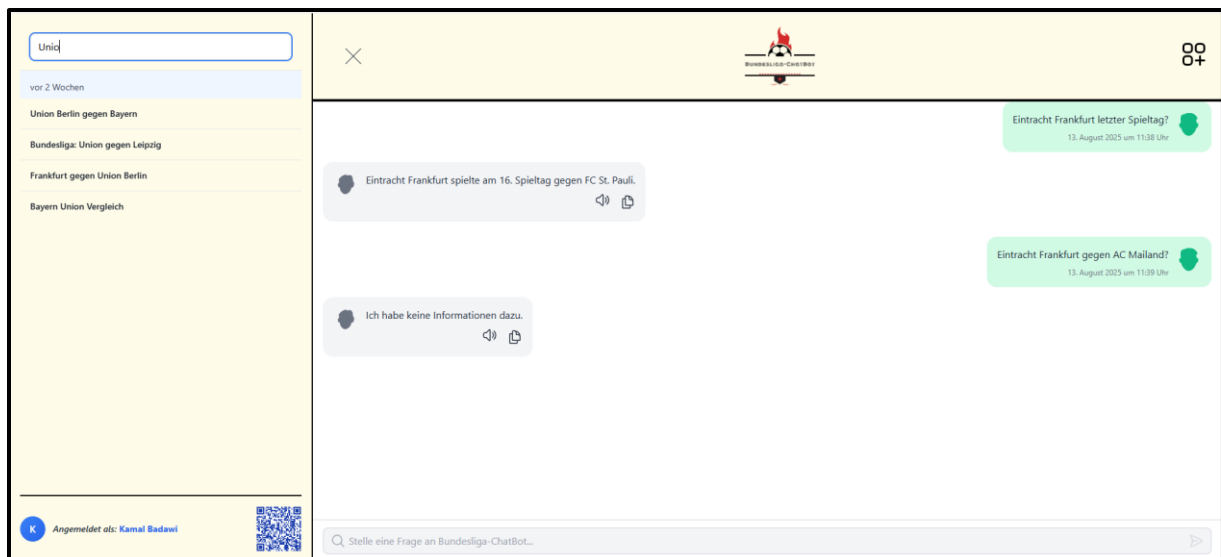
Wenn eine Nutzerfrage nicht durch die Daten der *OpenLigaDB*-API abgedeckt ist, gibt der Bundesliga-ChatBot die Antwort: „Ich habe keine Informationen dazu“



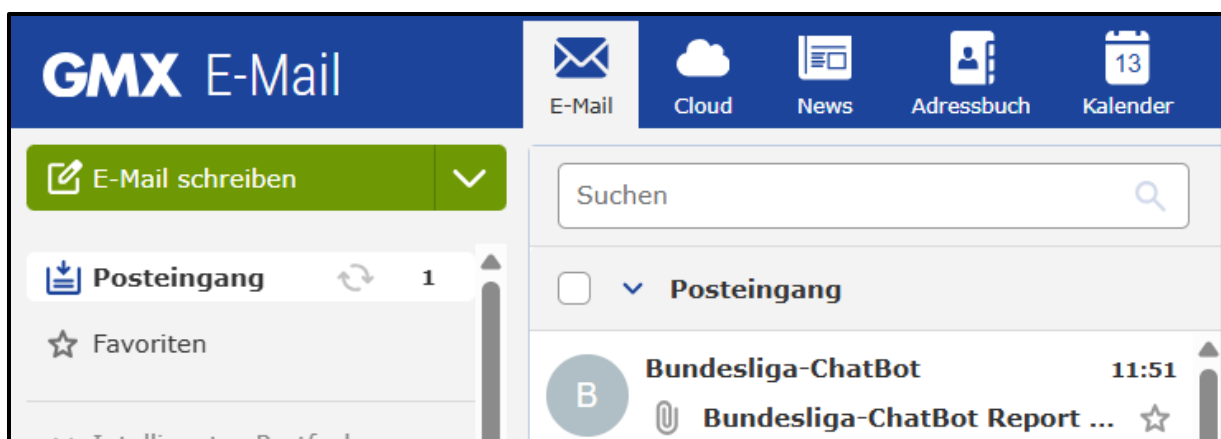
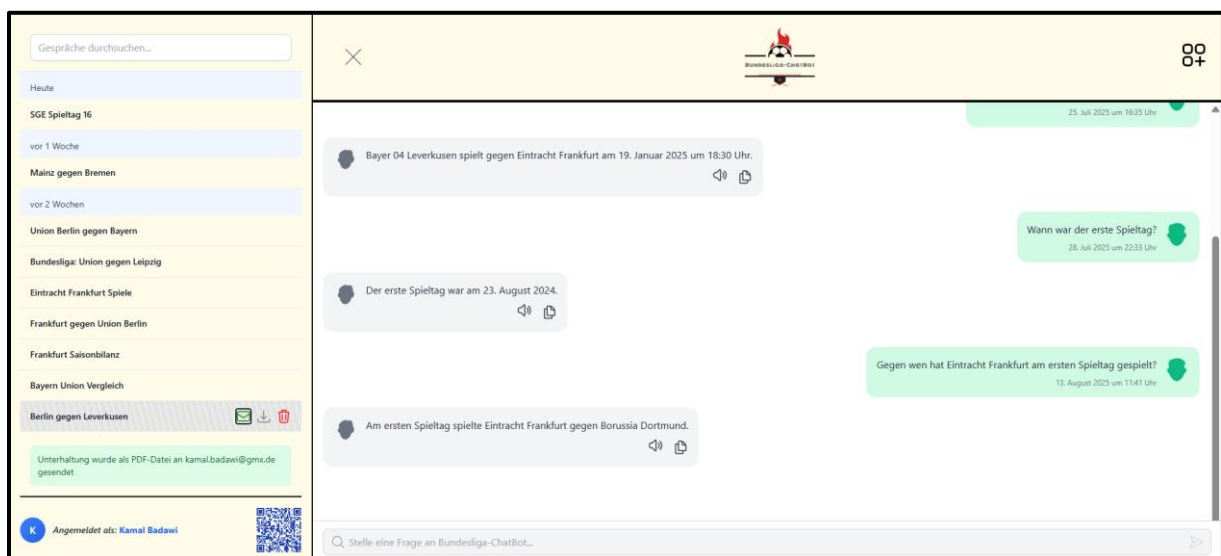
Wird ein älterer Chat aus dem Verlauf ausgewählt, wird der zugehörige Dialog geladen. Neue Fragen und Antworten werden diesem bestehenden Chat hinzugefügt - ohne eine neue *conversation\_id* zu erzeugen. Der ursprüngliche Titel bleibt dabei erhalten. Die neuen Einträge, inklusive aller relevanten Metadaten (wie Datum und Uhrzeit), werden dennoch in der *MongoDB* gespeichert.



Auf der linken Seite befindet sich ein Bereich, in dem der Nutzer seine bisherigen Chatverläufe einsehen kann - absteigend sortiert nach Datum und Uhrzeit. Zusätzlich besteht die Möglichkeit, den Verlauf nach einem bestimmten Text oder Unterhaltungstitel zu filtern, um einen schnellen Zugriff auf relevante Unterhaltungen zu ermöglichen. Außerdem sieht man, wer gerade angemeldet ist. Über den QR-Code gelangt man zum Bundesliga-ChatBot, den man mit anderen Fans teilen kann.



Außerdem hat der Nutzer die Möglichkeit, eine Unterhaltung per E-Mail als PDF-Datei an seine E-Mail-Adresse zu senden - derzeit über *Gmail*, zukünftig über *SendGrid*. Zusätzlich kann der Nutzer die Unterhaltung lokal als PDF-Datei speichern oder sie löschen.



# Bundesliga-ChatBot Report vom 13-08-2025-11-51-53



**Bundesliga-ChatBot**

Details

13.08.2025 - 11:51



1 Anhang



[Mehr Speicherplatz für Anhänge](#)

Liebe Kundin, lieber Kunde,

anbei übersenden wir Ihnen die Unterhaltung.

Vielen Dank für die Nutzung des Bundesliga-ChatBots.

Mit freundlichen Grüßen

Ihr Bundesliga-ChatBot Team

Sofortantwort hier schreiben...

Sofort antworten



Liebe Kundin, lieber Kunde,

anbei übersenden wir Ihnen die Unterhaltung

---

**Berlin gegen Leverkusen**

Basierend auf den vorliegenden Daten gab es ein Spiel zwischen 1. FC Union Berlin und Bayer 04 Leverkusen am Spieltag 12, welches Leverkusen mit 2:1 gewann.

**Byern Lverkusen gegen Eintracht? wann?**

Bayer 04 Leverkusen spielt gegen Eintracht Frankfurt am 19. Januar 2025 um 18:30 Uhr.

**Wann war der erste Spieltag?**

Der erste Spieltag war am 23. August 2024.

**Gegen wen hat Eintracht Frankfurt am ersten Spieltag gespielt?**

Am ersten Spieltag spielte Eintracht Frankfurt gegen Borussia Dortmund.

---

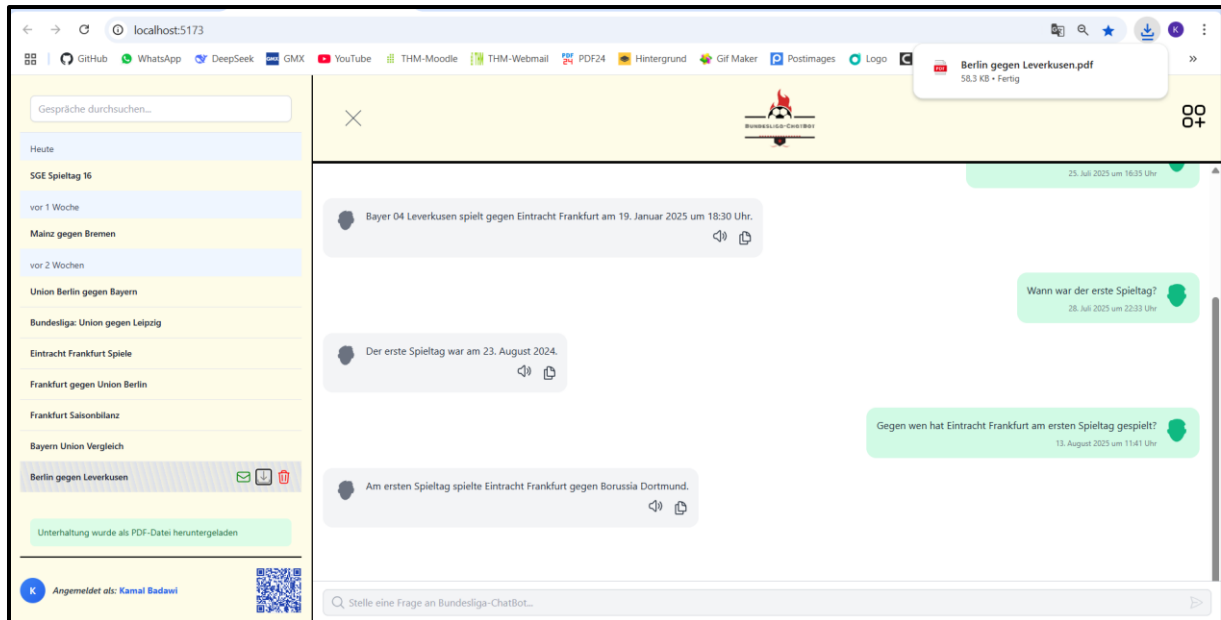
Vielen Dank für die Nutzung des Bundesliga-ChatBots.

Mit freundlichen Grüßen,

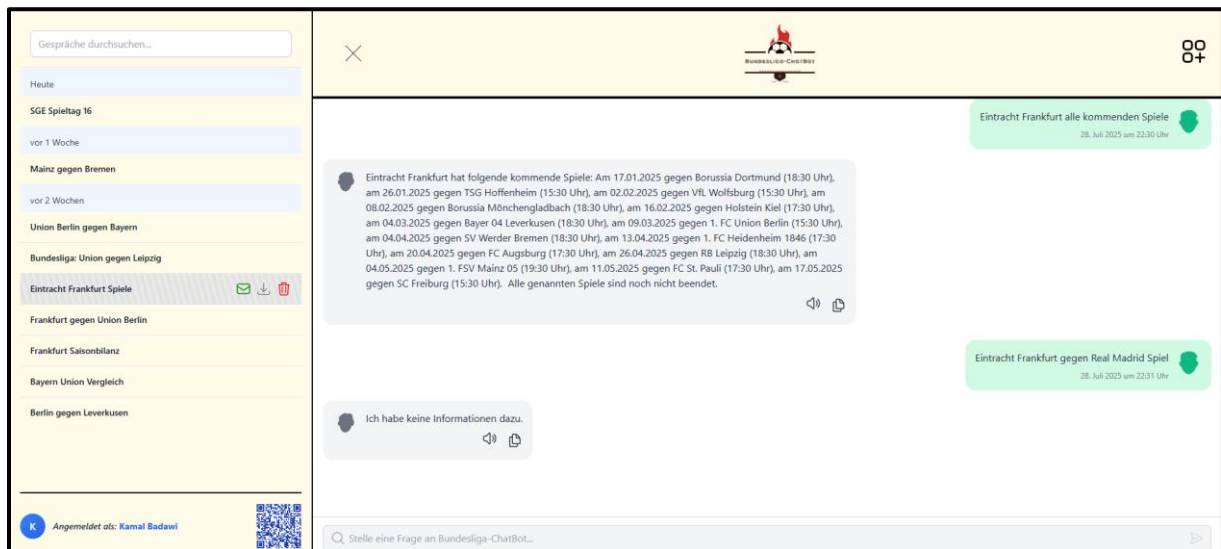
Ihr Bundesliga-ChatBot Team

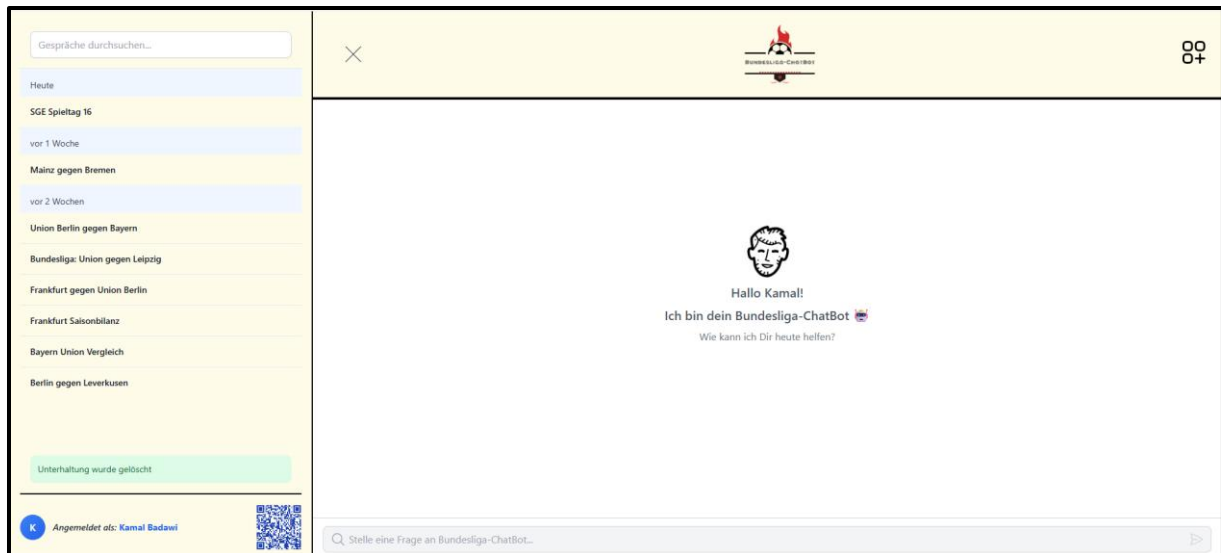
*Bundesliga-ChatBot*

Erstellt am 13. 08. 2025 um 11:51:53

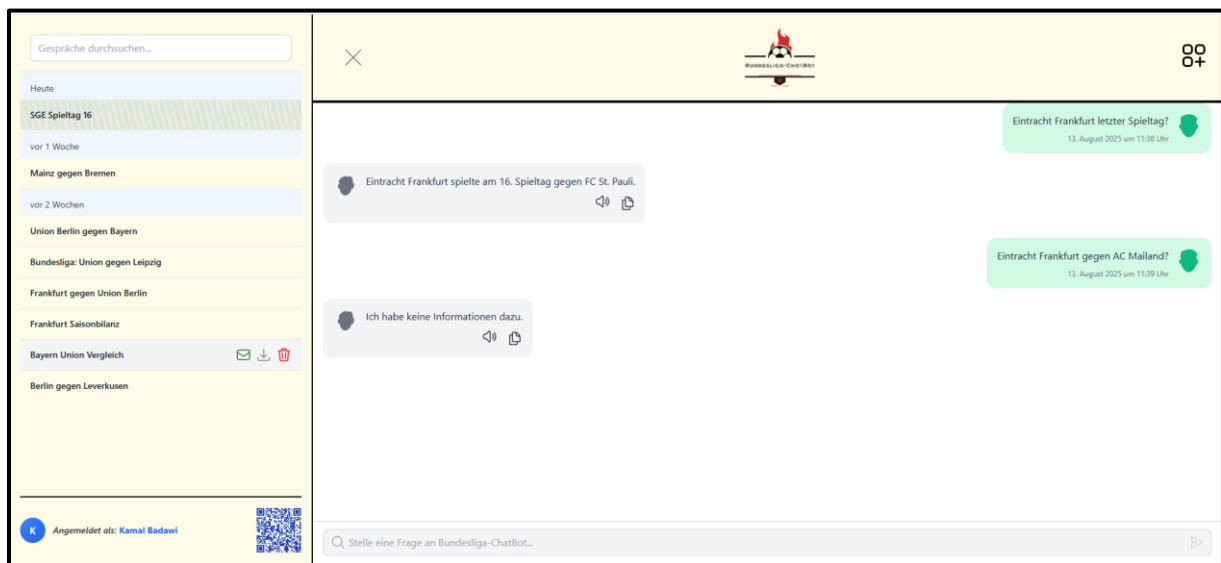


Wird eine Unterhaltung gelöscht, in der sich der Nutzer aktuell befindet, wird diese vollständig entfernt und automatisch eine neue Unterhaltung mit einer neuen `conversation_id` gestartet.

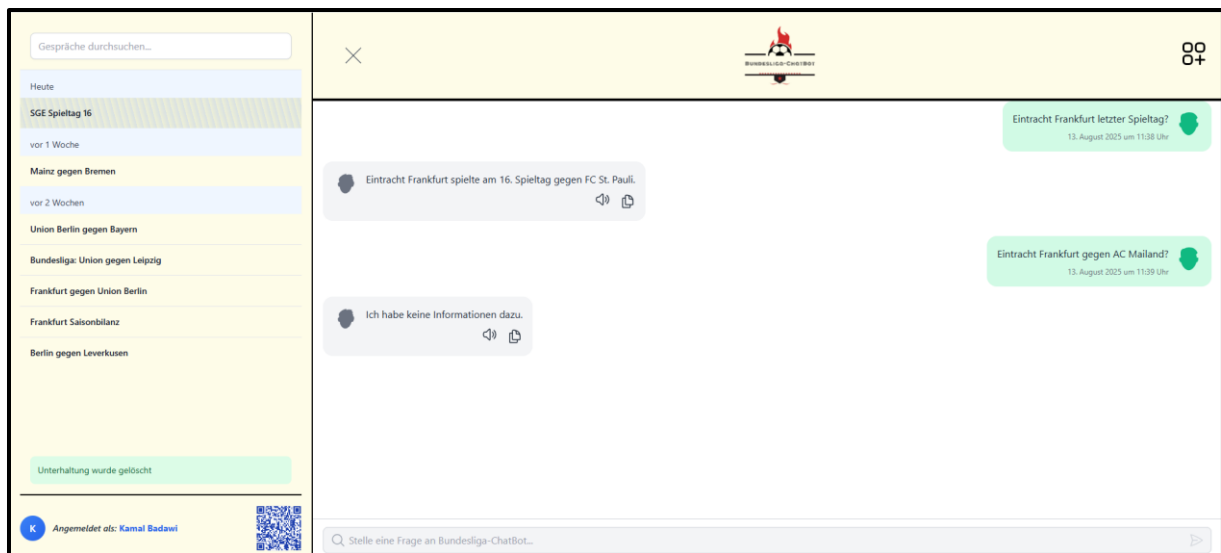




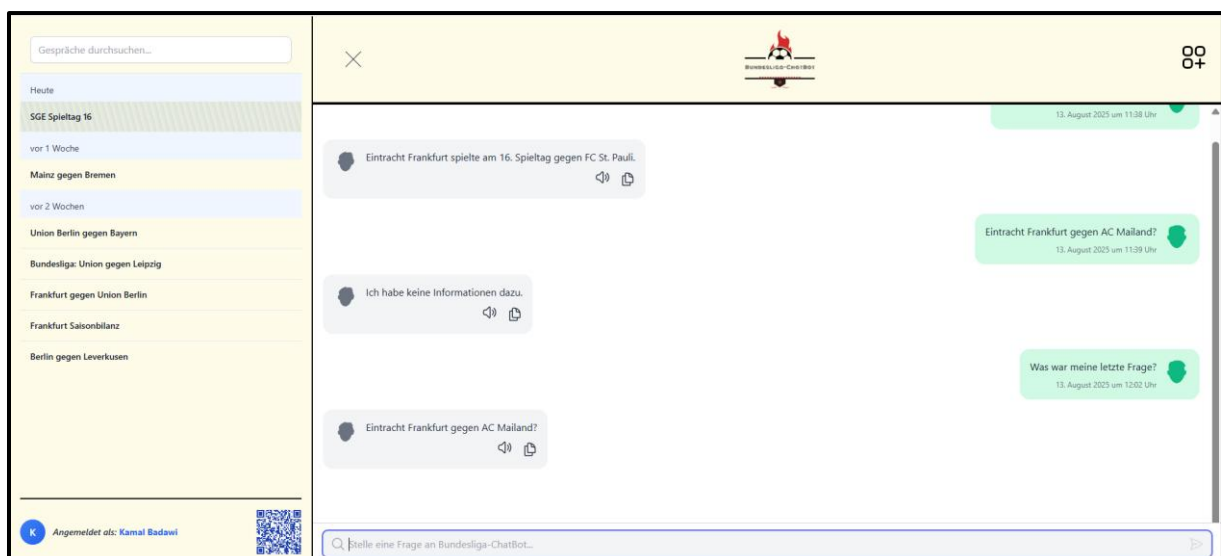
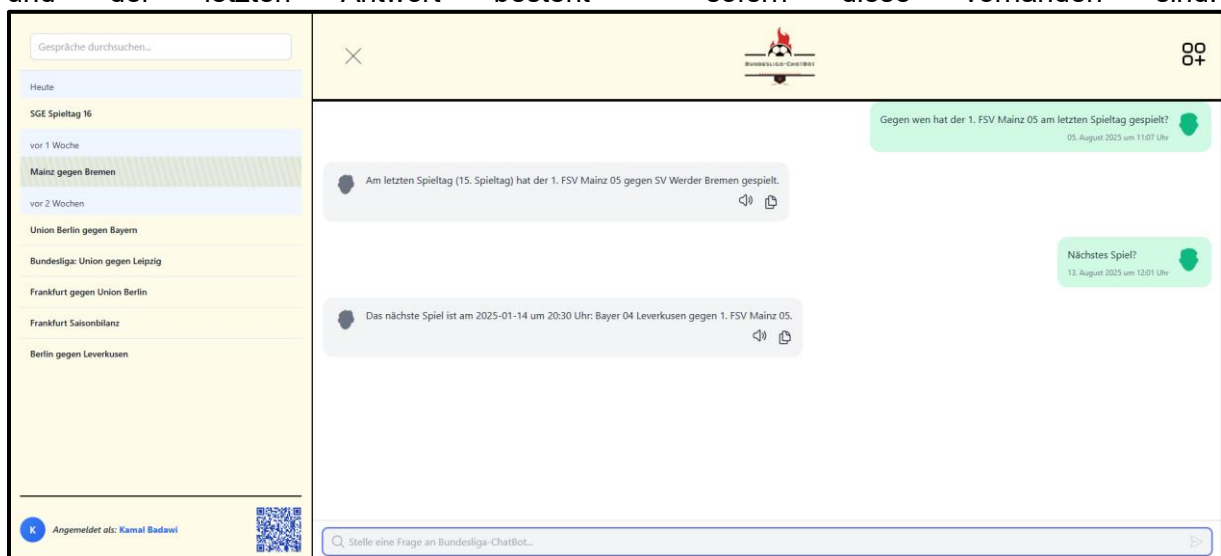
Wenn hingegen eine andere (nicht aktive) Unterhaltung gelöscht wird, wird diese lediglich aus der *MongoDB* entfernt, ohne eine neue Unterhaltung zu beginnen. Der Nutzer bleibt dabei in seiner aktuellen Unterhaltung.







Der Bundesliga-ChatBot verfügt über ein kurzfristiges Gedächtnis, das aus der letzten Frage und der letzten Antwort besteht - sofern diese vorhanden sind.



## 8. Test-Szenarien und Teststrategie

### 8.1. Frontend Tests (*React + TailwindCSS*)

#### 8.1.1 Initiales Laden

- ✚ **Erwartung:** UI lädt leer, User-Info sichtbar, Chatliste geladen.
- ✚ **Tools:**
  - *Jest (Unit Tests)*
  - *React Testing Library (DOM Rendering)*

#### 8.1.2 Frage stellen

- ✚ **Erwartung:** Eingabe akzeptiert, „Bundesliga-ChatBot denkt ...“-Animation erscheint, Antwort sichtbar.
- ✚ **Tools:**
  - *React Testing Library (Input + Events)*
  - *Cypress / Playwright (E2E Browser Test)*

#### 8.1.3 Neuer Chat starten

- ✚ **Erwartung:** Chat-Bereich wird geleert, neue `conversation_id` erstellt.
- ✚ **Tools:**
  - *Jest (State-Management Test)*
  - *Cypress (User Flow Test)*

#### 8.1.4 Chatverlauf öffnen

- ✚ **Erwartung:** Alte Nachrichten laden, neue Nachrichten anhängen.
- ✚ **Tools:**
  - *React Testing Library*
  - *Cypress (Navigation im UI)*

#### 8.1.5 Chat durchsuchen/filtern

- ✚ **Erwartung:** Nur relevante Chats (Titel) werden angezeigt.
- ✚ **Tools:**
  - *Jest (Filter-Funktion Unit Test)*
  - *Cypress (UI Suche/Filter)*

#### 8.1.6 Chat löschen (aktiver Chat)

- ✚ **Erwartung:** Aktueller Chat wird entfernt → neuer Chat gestartet.
- ✚ **Tools:**
  - *Cypress (E2E Klick-Test)*
  - *React Testing Library (State Reset prüfen)*

### 8.1.7 Chat löschen (nicht-aktiver Chat)

✚ **Erwartung:** Nur dieser Chat verschwindet, aktueller bleibt erhalten.

✚ **Tools:**

- *Cypress (Listen-Interaktion)*
- *React Testing Library*

### 8.1.8 Export als PDF (lokal speichern)

✚ **Erwartung:** PDF wird generiert und heruntergeladen.

✚ **Tools:**

- *Jest (Button Event Unit Test)*
- *Cypress (Download testen)*
- *reportlab (Backend PDF-Erstellung)*

### 8.1.9 Chat per E-Mail senden (*Gmail/SendGrid*)

✚ **Erwartung:** PDF wird erstellt und per Mail verschickt.

✚ **Tools:**

- *Cypress (UI Button + Bestätigung)*
- *pytest + smtplib (Gmail Test)*
- *pytest + httpx + responses (SendGrid Mock API)*

### 8.1.10 Responsives Layout / Mobile View

✚ **Erwartung:** App funktioniert auf Desktop, Tablet, Mobile.

✚ **Tools:**

- *Cypress (Viewport-Simulation)*
- *Percy / Chromatic (Visual Regression)*

## 8.2 Backend Tests (FastAPI, LangChain, MongoDB, SQLite)

### 8.2.1 API `/question`

✚ **Erwartung:** Anfrage wird korrekt verarbeitet, Antwort aus *Gemini 2.5 Pro/OpenLigaDB* zurückgegeben.

✚ **Tools:**

- *pytest*
- *FastAPI TestClient*
- *httpx (async Client Tests)*

### 8.2.2 API `/conversations_info`

✚ **Erwartung:** Gibt Konversationen sortiert zurück.

✚ **Tools:**

- *pytest*
- *mongomock (MongoDB Mocking)*

### 8.2.3 Speicherung in *MongoDB*

✚ **Erwartung:** Chat inkl. Metadaten wird gespeichert.

✚ **Tools:**

- *pytest*
- *mongomock*

### 8.2.4 *SQLite* Fallback

✚ **Erwartung:** Bei *OpenLigaDB*-Ausfall werden *SQLite*-Daten genutzt.

✚ **Tools:**

- *pytest*
- *SQLite In-Memory Mode*

### 8.2.5 RAG mit *Langchain*

✚ **Erwartung:** Embeddings werden genutzt, Kontext wird korrekt hinzugefügt.

✚ **Tools:**

- *pytest*
- *langchain.testing utilities (Retriever/Embedding Mocks)*

### 8.2.6 PDF-Erstellung Backend

✚ **Erwartung:** PDF wird mit Chat-Inhalten generiert.

✚ **Tools:**

- *pytest*
- *reportlab*

### 8.2.7 Mailversand *Gmail*

✚ **Erwartung:** PDF wird über *Gmail* versendet.

✚ **Tools:**

- *pytest*
- *smtplib*
- *pytest-mock (Credential Mocking)*

### 8.2.8 Mailversand *SendGrid*

✚ **Erwartung:** PDF wird über *SendGrid*-API verschickt.

✚ **Tools:**

- *pytest*
- *httpx*
- *responses (HTTP Mocking)*

## 8.3. Datenbank Tests (*MongoDB* + *SQLite*)

### 8.3.1 *MongoDB* Chat-Verwaltung

✚ **Erwartung:** Speichern, Laden, Löschen von Konversationen funktioniert.

✚ **Tools:**

- *pytest*
- *mongomock*

### 8.3.2 *SQLite OpenLigaDB* Datenhaltung

✚ **Erwartung:** Daten werden persistiert und bei Bedarf als Fallback genutzt.

✚ **Tools:**

- *pytest*
- *SQLite In-Memory Mode*

### 8.3.3 Konsistenzprüfung *MongoDB/SQLite*

✚ **Erwartung:** Chats und Spielpläne passen zusammen.

✚ **Tools:**

- *pytest*
- *faker*

### 8.3.4 Filter-Queries in *MongoDB*

✚ **Erwartung:** Titel-Suche liefert korrekte Chats.

✚ **Tools:**

- *pytest*
- *mongomock*

## 8.4. End-to-End (System + User Flows)

### 8.4.1 Vollständige Session

✚ **Ablauf:** Neuer Chat → Frage → alter Chat öffnen → Export als PDF.

✚ **Erwartung:** Alles konsistent, `conversation_id` korrekt.

✚ **Tools:**

- *Cypress*
- *Playwright*

### 8.4.2 Fehlerhandling (API Key ungültig)

✚ **Erwartung:** Sinnvolle Fehlermeldung statt Crash.

✚ **Tools:**

- *pytest*
- *Cypress*

### 8.4.3 Fehlerhandling (*OpenLigaDB* Timeout → *SQLite*)

✚ **Erwartung:** Fallback *SQLite*-Daten werden genutzt.

✚ **Tools:**

- *pytest*
- *Cypress*

### 8.4.4 Fehlerhandling (Datenbank nicht erreichbar)

✚ **Erwartung:** Meldung „Datenbank nicht erreichbar“.

✚ **Tools:**

- *pytest*
- *Cypress*

### 8.4.5 Löschlogik aktiv/nicht-aktiv

✚ **Erwartung:** Verhalten wie spezifiziert (Neustart bei aktivem Chat, nur Entfernen bei inaktivem Chat).

✚ **Tools:**

- *Cypress*
- *React Testing Library*

## 8.5. Nicht-funktionale Tests

### 8.5.1 Performance (100 gleichzeitige User)

✚ **Erwartung:** Antwortzeit < 3 Sekunden.

✚ **Tools:**

- *Locust*
- *k6*

### 8.5.2 Sicherheit: *SQL Injection*

✚ **Erwartung:** Keine Manipulation möglich.

✚ **Tools:**

- *pytest*
- *OWASP ZAP*

### 8.5.3 Sicherheit: *XSS* im Frontend

✚ **Erwartung:** Script-Tag wird escaped.

✚ **Tools:**

- *jest-axe*
- *Cypress*

#### 8.5.4 Zugriffsschutz `conversation_id` & `user_id`

✚ **Erwartung:** User darf nur eigene Chats öffnen.

✚ **Tools:**

- *pytest*
- *httplib*

#### 8.5.5 Netzwerkunterbrechung

✚ **Erwartung:** Meldung „Verbindung unterbrochen“.

✚ **Tools:**

- *Cypress*
- *Playwright*

#### 8.5.6 Usability (Mobile/Desktop)

✚ **Erwartung:** Layout passt sich an, Bedienbarkeit bleibt erhalten.

✚ **Tools:**

- *Cypress (Viewport-Tests)*
- *Percy / Chromatic (Visual Regression)*

## 9. Authentifizierung und Zugriffskontrolle

Die Authentifizierung des Bundesliga-ChatBots erfolgt über eine Login-Maske, in der sich Nutzer mit Benutzernamen oder E-Mail-Adresse und Passwort anmelden. Alternativ kann optional auch eine Anmeldung über OAuth-Dienste wie *Google* oder *GitHub* ermöglicht werden. Um die Sicherheit der Anmeldedaten zu gewährleisten, werden Passwörter grundsätzlich mit einem sicheren Verfahren wie *bcrypt* gehasht und niemals im Klartext gespeichert. Sämtliche Verbindungen erfolgen verschlüsselt über *HTTPS*. Zusätzlich wird ein Rate-Limiting für Login-Versuche implementiert, um *Brute-Force-Angriffe* zu verhindern. Die technische Umsetzung der Authentifizierung basiert auf *FastAPI* in Kombination mit *fastapi-users* oder der Nutzung von *JWTs* über *pyjwt*. Auf der Client-Seite übernimmt das *React*-Frontend mit *TailwindCSS* die Validierung der Formulareingaben. Die Absicherung der Authentifizierung wird durch automatisierte Tests mit *pytest* (Backend), *Cypress* (UI-Tests für Login) sowie *Postman* (API-Auth-Tests) überprüft.

Die Zugriffskontrolle erfolgt nach erfolgreichem Login durch die Ausstellung eines *JWT*-Tokens, der bei jedem weiteren API-Aufruf geprüft wird. Dadurch wird sichergestellt, dass nur authentifizierte Nutzer Zugriff auf die geschützten Funktionen und Daten erhalten. Jeder Nutzer kann ausschließlich seine eigenen Inhalte einsehen, darunter Chatverläufe, PDF-Exporte oder per E-Mail versendete Unterhaltungen. Hierfür wird bei jeder Anfrage die jeweilige `user_id` serverseitig überprüft, sodass ein Zugriff auf fremde `conversation_id` ausgeschlossen ist. Optional kann zudem ein Rollenmodell implementiert werden, um beispielsweise administrative Zugriffsrechte zu ermöglichen. Die technische Umsetzung der Zugriffskontrolle nutzt *FastAPI Dependency Injection* zur Validierung des aktuellen Nutzers, während auf Datenbankebene Filter in *MongoDB* (`find({ user_id: current_user.id })`) sicherstellen, dass nur eigene Datensätze geladen werden. Die Tests zur Zugriffssicherheit erfolgen durch *pytest* in Kombination mit *mongomock* für Datenbanktests.

sowie durch *Cypress* für End-to-End-Szenarien, etwa die Prüfung, dass ein Nutzer keine fremden Chats öffnen kann.

## 10. Versionskontrollsystem (*Git* + *GitHub*)

Für die Entwicklung und Verwaltung des Bundesliga-ChatBots wird *Git* als Versionskontrollsystem eingesetzt. *Git* ermöglicht es, Änderungen am Quellcode nachzuvollziehen, verschiedene Entwicklungszweige (Branches) parallel zu pflegen und bei Bedarf ältere Versionen des Projekts wiederherzustellen. Dadurch wird eine strukturierte, transparente und nachvollziehbare Softwareentwicklung gewährleistet.

Zur zentralen Verwaltung und Zusammenarbeit wird *GitHub* genutzt. Dort ist das Projekt in einem Repository verfügbar, über das alle Teammitglieder Zugriff auf den aktuellen Entwicklungsstand haben. Über Pull Requests können Änderungen überprüft und diskutiert werden, bevor sie in den Hauptzweig integriert werden. Zudem bietet *GitHub* Funktionen wie Issue-Tracking, Projektboards und Continuous Integration Workflows, die den Entwicklungsprozess zusätzlich unterstützen. Damit steht jederzeit eine aktuelle und konsistente Version des Projekts zur Verfügung, die sowohl für die Weiterentwicklung als auch für die Qualitätssicherung genutzt werden kann.

Das Repository des Bundesliga-ChatBots ist derzeit öffentlich (Public) über den folgenden Link zugänglich: <https://github.com/kamal-badawi/Bundesliga-ChatBot-Gemini>

In Zukunft wird der Zugriff auf das Repository jedoch privat sein und nur noch über Einladung per Link oder E-Mail erfolgen, um die Vertraulichkeit des Projekts zu gewährleisten.

## 11. Fazit

Der Bundesliga-ChatBot bietet gegenüber allgemeinen KI-Assistenten wie *ChatGPT* oder *DeepSeek* zahlreiche Vorteile. Wie bei diesen Systemen können Nutzer Antworten unkompliziert kopieren oder sich vorlesen lassen, was die Bedienung besonders komfortabel gestaltet. Ein entscheidender Mehrwert des Bundesliga-ChatBots liegt jedoch in seiner direkten Anbindung an die *OpenLigaDB*-API. Dadurch hat er jederzeit Zugriff auf aktuelle und verlässliche Bundesliga-Daten. Fragen zu Spielständen, Tabellenplatzierungen, Spielerstatistiken oder kommenden Begegnungen werden so stets präzise und zeitnah beantwortet.

Darüber hinaus ist der Bundesliga-ChatBot gezielt auf die Bedürfnisse von Fußballfans zugeschnitten. Er versteht kontextspezifische Fragen zur Bundesliga deutlich besser als allgemeine KI-Modelle. Dank seines kurzfristigen Gedächtnisses kann er auf vorherige Fragen und Antworten Bezug nehmen, was einen flüssigen und zusammenhängenden Dialog ermöglicht. Dies sorgt für eine persönlichere und interaktivere Nutzererfahrung.

Praktische Funktionen zur Verwaltung und Speicherung von Unterhaltungen runden das Angebot ab. Nutzer können Chatverläufe speichern, exportieren - beispielsweise als PDF - oder Chats starten und löschen, um die Übersicht zu behalten. Diese Features erlauben eine individuell anpassbare Nutzung, erleichtern den Zugriff auf Informationen und machen die Interaktion noch angenehmer.



Insgesamt vereint der Bundesliga-ChatBot die Flexibilität und Intelligenz moderner KI-Technologie mit der Präzision und Aktualität speziell aufbereiteter Sportdaten. Diese Kombination macht ihn zu einem wertvollen Tool für Fans, Journalisten und alle, die sich für Fußball begeistern. Mit ihm wird das Verfolgen der Liga nicht nur einfacher, sondern auch informativer und interaktiver.

Zukünftig soll durch die Kombination aus redundanten Datenbanken, Caching-Mechanismen und RAG-Integration eine hohe Verfügbarkeit, schnelle Reaktionszeiten und qualitativ hochwertige, kontextbasierte Antworten sichergestellt werden. Während der Bundesliga-Spieltage werden die Daten automatisch im Drei-Sekunden-Takt aktualisiert, außerhalb der Spieltage pausiert die Pipeline, um Ressourcen zu schonen.

# Quellen

ChatGPT	<a href="https://chatgpt.com/">https://chatgpt.com/</a>
Fast API Logo	<a href="https://www.bing.com/images/search?q=fastapi%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=fastapi%20symbol&amp;sc=0-14&amp;cvid=B7DAE97C9474422282A166BAF59449E5&amp;first=1">https://www.bing.com/images/search?q=fastapi%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=fastapi%20symbol&amp;sc=0-14&amp;cvid=B7DAE97C9474422282A166BAF59449E5&amp;first=1</a>
Gemini 2.5 Pro Logo	<a href="https://www.bing.com/images/search?q=gemini+2.5+pro+symbol&amp;FORM=HDRSC3">https://www.bing.com/images/search?q=gemini+2.5+pro+symbol&amp;FORM=HDRSC3</a>
Github-Link (Code)	<a href="https://github.com/kamal-badawi/Bundesliga-ChatBot-Gemini">https://github.com/kamal-badawi/Bundesliga-ChatBot-Gemini</a>
MongoDB Logo	<a href="https://www.bing.com/images/search?q=mongodb%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=mongodb%20symbol&amp;sc=1-14&amp;cvid=F9A0624BEF0B4156A952A78A038BECFAF&amp;first=1">https://www.bing.com/images/search?q=mongodb%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=mongodb%20symbol&amp;sc=1-14&amp;cvid=F9A0624BEF0B4156A952A78A038BECFAF&amp;first=1</a>
OpenLigaDB Logo	<a href="https://www.bing.com/images/search?q=openligadb+logo+symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=openligadb+logo+symbol&amp;sc=1-22&amp;cvid=04D3B12BD7FC4423B6C86CC9B04FD8FC&amp;first=1">https://www.bing.com/images/search?q=openligadb+logo+symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=openligadb+logo+symbol&amp;sc=1-22&amp;cvid=04D3B12BD7FC4423B6C86CC9B04FD8FC&amp;first=1</a>
Python Logo	<a href="https://www.bing.com/images/search?q=python%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=python%20symbol&amp;sc=10-12&amp;cvid=F1FED564B468489991211A2D33EDB275&amp;first=1">https://www.bing.com/images/search?q=python%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=python%20symbol&amp;sc=10-12&amp;cvid=F1FED564B468489991211A2D33EDB275&amp;first=1</a>
React Logo	<a href="https://www.bing.com/images/search?q=react%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=react%20symbol&amp;sc=6-12&amp;cvid=85251005FD8046BB996ED0C4DF2BC31A&amp;first=1">https://www.bing.com/images/search?q=react%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=react%20symbol&amp;sc=6-12&amp;cvid=85251005FD8046BB996ED0C4DF2BC31A&amp;first=1</a>
SQLite3 Logo	<a href="https://www.bing.com/images/search?q=sqlite%203%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=sqlite%203%20symbol&amp;sc=0-15&amp;cvid=A00935613B9440F0BED7D4BD46AFA86B&amp;first=1">https://www.bing.com/images/search?q=sqlite%203%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=sqlite%203%20symbol&amp;sc=0-15&amp;cvid=A00935613B9440F0BED7D4BD46AFA86B&amp;first=1</a>
TailwindCSS Logo	<a href="https://www.bing.com/images/search?q=tailwind%20css%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=tailwind%20css%20symbol&amp;sc=1-18&amp;cvid=1BFC1FFB226F437A864A7FB204EFB7F8&amp;first=1">https://www.bing.com/images/search?q=tailwind%20css%20symbol&amp;qs=n&amp;form=QBIR&amp;sp=-1&amp;lq=0&amp;pq=tailwind%20css%20symbol&amp;sc=1-18&amp;cvid=1BFC1FFB226F437A864A7FB204EFB7F8&amp;first=1</a>
Template-Vorlage	<a href="https://poweredtemplate.com/de/presentation-templates/bundesliga/license=free;type=powerpoint">https://poweredtemplate.com/de/presentation-templates/bundesliga/license=free;type=powerpoint</a>