Cloud and Distributed Computing Project

Software Architecture Documentation

“DigiBrain”

Inhaltsverzeichnis

1 Einführung und Ziele

Autor(en): NM, NO

Entwickler: NM, EP, NO

Die vorliegende Softwarearchitekturbeschreibung widmet sich der Web-Anwendung „DigiBrain“. Die Einleitung dient als Ausgangspunkt für die weiterführende Betrachtung der Softwarearchitektur gemäß den Vorgaben des arc42-Templates.

Das Projekt ist im Rahmen des Moduls "Cloud and Distributed Computing" entstanden. Die wesentliche Aufgabe bestand darin, eine Web-Anwendung zu entwickeln und in einer Gruppe umzusetzen. Dabei sollen moderne Tools und Frameworks verwendet werden, die im Verlauf der Vorlesung behandelt wurden.

DigiBrain ist eine Anwendung zum Brainstorming und Managen von Aufgaben oder Ideen. Die Browseranwendung ist zur gleichzeitigen Nutzung von einem Team ausgelegt. Nutzer können in einem gemeinsam nutzbaren Raum Themengebiete anlegen. Zu diesen Themen können die Nutzer dann wiederum Karten anlegen, die andere Mitglieder des Raumes betrachten können. Zur Unterstützung der Nutzer wird GPT zum einen das Brainstorming unterstützen, sowie als direkte Wissensquelle dienen.

1.1 Anforderungsbeschreibung

Bei der Planung des Projekts und in den darauffolgenden Sprint Plannings haben sich folgende funktionale Anforderungen an die Anwendung ergeben:

Bereich	Anforderung
Authentifizierung	Das Registrieren eines Accounts mit Nutzername und Passwort
	Anmelden in einen Account mit Nutzername und Passwort
Raumübersicht	Alle beigetretenen Räume werden in einer Liste angezeigt
	Einem Raum mit Raum-ID beitreten
	Einen beigetretenen Raum betreten
	Einen Raum mit Keywords erstellen
	Einen Raum löschen (Nur wenn man ihn auch erstellt hat)
	Einen Raum verlassen (Nur wenn man ihn nicht erstellt hat)

Raum	Ein Raum mit Sections und zugehörigen Karten wird angezeigt. Auch wird eine Schnittstelle zu GPT geboten, sowie Informationen zum Raum angezeigt.
	Eine Section mit einem Namen anlegen
	Eine Karte unter einer Section anlegen. Diese beinhaltet Überschrift, Labels und Text und hat eine auswählbare Farbe.
	Eine Karte bearbeiten (Nur wenn man sie angelegt hat)
	Eine Karte löschen (Nur wenn man sie angelegt hat)
	Den Zustand eines Raumes zu einer bestimmten Zeit unter einem Label speichern
	Einen Raum auf einen gespeicherten Zustand setzen
	Gespeicherte Zustände nach einem Substring durchsuchen und Ergebnisse nach Datum oder nach Treffern sortieren.
	Raum nach Substring filtern: Nur Karten/Sections anzeigen, die den Substring beinhalten.
	GPT über Keywords des Raums brainstormen lassen
	Fragen an GPT stellen
	Zurück zur Raumübersicht

1.2 Zielbeschreibung

Zur Messung der technischen Qualität der Anwendung wurden zu Beginn weitere Ziele definiert, die sich während der Entwicklung verfeinert haben. Dafür richteten wir uns an die Norm für Qualitätskriterien und Bewertung von System und Softwareprodukten – ISO 25010.

Qualitätskategorie	Qualitätskriterium	Beschreibung
Funktionalität	Angemessene Funktionalität	Die nötige Funktionalität wurde zu Beginn des Projekts festgelegt und in den Sprint Plannings ggf. erweitert.
	Vollständigkeit	Die Anwendung ist gemäß den vorher festgelegten Anforderungen komplett funktional.
	Korrektheit	Alle geforderten Funktionalitäten sind korrekt umgesetzt.
Performance	Zeitverhalten	Die Anwendung reagiert sofort auf Benutzereingaben, selbst wenn die Serverantwort noch erwartet wird.
	Kapazität	Die Anwendung stellt die erforderliche Funktionalität unter Berücksichtigung ändernder Anforderungen dar, indem sie flexibel skalierbar ist.
	Effizienz	Die Kommunikationsdichte, sowie die Größe einzelner Kommunikationsnachrichten wird so klein wie möglich gehalten. Globale Änderungen werden nur bei Bedarf aktualisiert.
Sicherheit		Daten eines Nutzers können nur angefragt werden, wenn dieser auch authentifiziert ist.
Kompatibilität		Die Anwendung ist unabhängig von Gerät (Mobile, Desktop) und Betriebssystem voll funktional und nutzbar.
Verlässlichkeit	Verfügbarkeit	Die Anwendung ist immer verfügbar, wenn

		eine Internetverbindung besteht.
	Fehlertoleranz	Fehlerhafte Eingaben werden abgefangen und falsche Anfragen an das Backend verhindert.
	Wiederherstellbarkeit	Durch das vollständige cloud basierte hosting der App werden ausgefallene Nodes automatisch wiederhergestellt.
Usability		Die Anwendung bietet durch ein intuitives, simples Userinterface eine gute Bedienbarkeit, einen schnellen und leichten Lernprozess und ein schönes, kreatives Design.
Wartung	Modularität	Die Anwendung bietet durch ihre Micro-Service Architektur eine erleichterte Wartung und gute Testmöglichkeiten einzelner Services.
	Wiederverwendbare Komponenten	Die Anwendung ist mit wiederverwendbaren Komponenten gebaut.
Portierbarkeit		Die Anwendung ist als Web-Anwendung von Überall nutzbar ohne sie zuerst installieren zu müssen.

2 Randbedingungen

Autor(en): NM

Die Randbedingungen der Aufgabenstellung werden in technische und organisatorische Randbedingungen geteilt.

2.1 Technische Randbedingungen

Die Anwendung soll eine Web-Anwendung im Micro-Service Architekturstil sein. Jeder Service soll dabei in einem eigenen Docker Container laufen. Die Datenbanken können entweder im gleichen oder in separaten Containern sein.

Für das Frontend stehen folgende Frameworks/Bibliotheken zur Auswahl: React, Angular, Vue, Svelte oder Ionic (Oder andere nach Rücksprache). Im Frontend soll besonderen Wert auf die Responsiveness und Kompatibilität mit sowohl Mobile Browsern als auch Desktop Browsern gelegt werden. Ebenso wichtig ist die User Experience.

Es soll ein Publish-Subscribe-System eingebaut werden, um Echtzeitdaten zu empfangen.

Als Backend-Optionen stehen zur Auswahl: Node.js/Express, LoopBack, Python/Falcon, Flask, Django REST Framework, Spring Boot, ktor oder Go (Oder andere nach Rücksprache). Um das Backend zu testen und dokumentieren, sollen Tools wie Swagger oder apiDoc verwendet werden.

Bei der Datenspeicherung wird eine NoSQL-Datenbank empfohlen, vorzugsweise in Kombination mit einer SQL-Datenbank (hybride Lösung). Die Verwendung von NoSQL sollte sich anhand der Anforderungen und Szenarien begründen lassen.

Die Bereitstellung der Anwendung soll in der Cloud mit containerisierten Backend-Micro-Servicen erfolgen. Als kostenfreie Hosting-Optionen werden bwCloud oder Azure empfohlen. Falls eine andere Hosting-Option verwendet werden soll, sollte man dies im vorher abstimmen.

Die Interfaces und Kommentare im Sourcecode sollen in Englisch formuliert sein.

2.2 Organisatorische Randbedingungen

Die Planung und Entwicklung des Projekts erfolgen mithilfe agiler Arbeitsmethoden ähnlich dem Scrum-Framework. Das Projekt wird in Sprints unterteilt. Zu jedem Sprint gehört ein Sprint Planning, die Entwicklungsphase und ein Sprint Review. Am Ende eines Sprints gibt es vordefinierte Milestones. In diesen werden mit dem Dozenten als Product Owner das Sprint Review des vorherigen Sprints abgehalten, sowie das Sprint Planning für den kommenden Sprint. Die Milestones sind wie folgt definiert:

- Exploratory MS
    - Genauere Vorstellung der Anforderungen an Funktionalität und Technik
    - Wahl der Technologien und erste Demonstrationen mit Diesen
- Architectural Spike MS
    - Informations-Architektur der Anwendung
    - Architektur-Diagramm der Anwendung (Komponenten und Kommunikation dieser)
    - Wireframe der UI
- Alpha MS
    - Kernfunktionalität implementiert
- Beta MS
    - Alle Funktionalitäten implementiert
- Certification MS
    - Anwendung voll funktional, getestet und performant
    - Fertige UI
    - Dokumentation
    - Präsentation der Anwendung
- Submission MS
    - Abgabe des zertifizierten Projekts

Für das Managen der Sprints soll Trello verwendet werden. Dieses soll mindestens die Artefakte „Product Backlog“, „Sprint Backlog“, „In Progress“, „Done 1…5“, „Impediments“ und „Risks“ enthalten und regelmäßig aktualisiert werden.

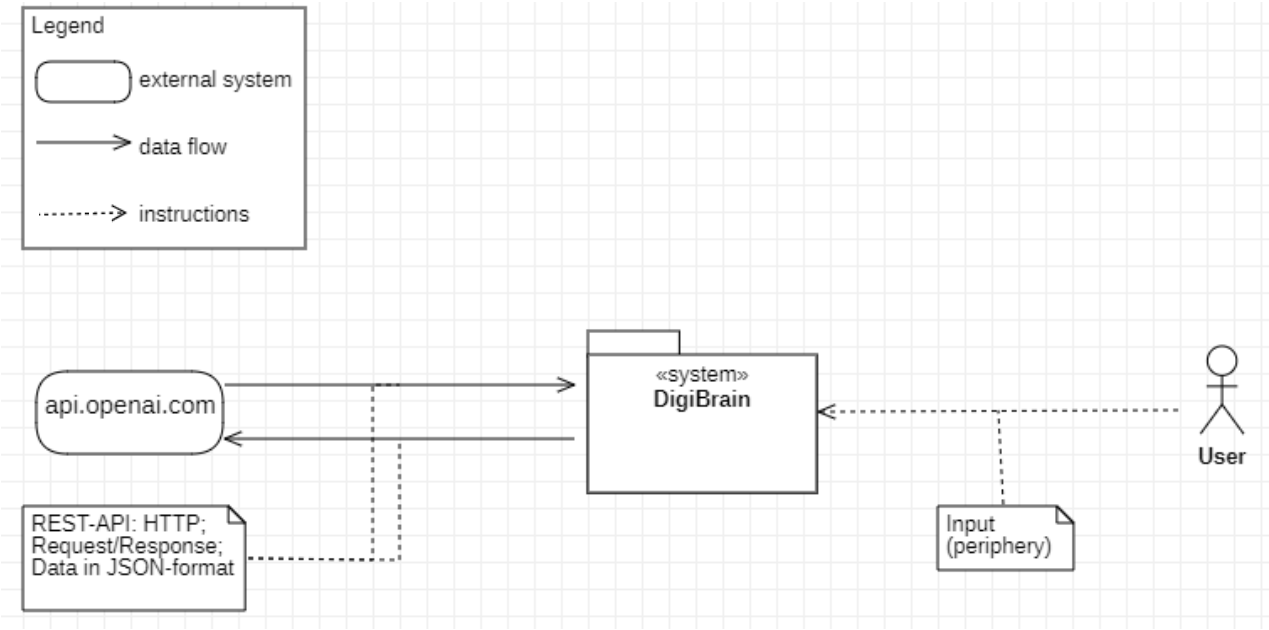
Als Entwicklungs-Repository ist Bitbucket vorgegeben. Dieses soll immer auf dem neuesten Stand sein.

3 Kontextabgrenzung

Autor(en): NM

Entwickler: NM, EP, NO

Das folgende Kontextdiagramm zeigt die Systemgrenzen von DigiBrain und dessen externe Schnittstellen bzw. Akteuren. Es zeigt somit die Verantwortlichkeit (Scope) von DigiBrain, sowie die die Verantwortung, die Nachbarsysteme übernehmen.



3.1 Akteure

3.1.1 api.openai.com

Die API von OpenAI stellt sowohl die Funktionalität für das KI-unterstützte Brainstorming als auch die Funktionalität für das KI-unterstützte Beantworten einer Frage. Somit ist DigiBrain selbst nicht für das Erzeugen der Daten verantwortlich, sondern nur für das Anfordern dieser.

3.1.2 Nutzer (User)

Der Nutzer bedient die App mittels Eingaben auf einem Gerät, welches Zugriff zu einem Browser und dem Internet hat. Somit ist DigiBrain selbst nicht zuständig für die Eingaben/Instruktionen, sondern nur für das korrekte Verarbeiten dieser.

3.2 Technischer Kontext

3.2.1 api.openai.com

Für beide Funktionalitäten, die DigiBrain über die OpenAI API anbietet, wird dieselbe API-Funktion genutzt.

In node.js wird die „create chat completion“-Funktion aufgerufen:

```
const openAi = new OpenAIApi(
  new Configuration({
    apiKey: process.env.OPENAI_API_KEY,
  }),
);
const response = await openAi.createChatCompletion({
  model: 'gpt-3.5-turbo',
  messages: [{role: 'user', content: prompt}],
});
```

Dies löst eine HTTPS POST-Request an `https://api.openai.com/v1/chat/completions` mit folgen Parametern aus:


```
{
 "model": "gpt-3.5-turbo",
 "messages": [{"role": "system", "content": "You are a helpful assistant."}, {"role": "user", "content": "Hello!"}]
}
```


Die Response-Nachricht enthält dann Daten in diesem Format:


```
{
 "id": "chatcmpl-123",
 "object": "chat.completion",
 "created": 1677652288,
 "choices": [{
  "index": 0,
  "message": {
   "role": "assistant",
   "content": "\n\nHello there, how may I assist you today?",
  },
  "finish_reason": "stop"
 }],
 "usage": {
  "prompt_tokens": 9,
  "completion_tokens": 12,
  "total_tokens": 21
 }
}
```

3.2.2 Nutzer (User)

Wenn der Nutzer die Anwendung über den Browser öffnet, bietet das Frontend die nötigen Komponenten, um definierte Aktionen auszulösen. Folgende Web-Komponenten bieten dem Nutzer dabei jeweilige Funktionalitäten.

Input-Elemente:

- Authentifizierung: Nutzername und Passwort eingeben
- Raum beitreten: Raum-ID eingeben
- Raum erstellen: Raum-Name und Keywords eingeben
- Section hinzufügen: Section-Name eingeben
- Karte hinzufügen/editieren: Karten-Header, -Labels und -Text eingeben
- Raum filtern: Filter-String eingeben
- Timestamp erstellen: Timestamp-Label eingeben
- Timestamps durchsuchen: Such-String eingeben
- Frage an GPT stellen: Prompt eingeben

Button- und Icon-Elemente:

- Authentifizierung: Nutzername und Passwort bestätigen, Ausloggen
- Raum beitreten: Popup aufrufen und Raum-ID bestätigen
- Raum erstellen: Popup aufrufen und Raum-Name und Keywords bestätigen
- Raum löschen, verlassen, betreten, austreten
- Section hinzufügen: Section-Name bestätigen
- Karte hinzufügen/editieren: „Editierkarte“ öffnen und Karten-Header, -Labels und -Text bestätigen/abbrechen
- Raum filtern: Kontextmenü öffnen
- Timestamp erstellen: Kontextmenü und Popup öffnen und Timestamp-Label bestätigen
- Timestamps durchsuchen: Kontextmenü und Popup öffnen Such-String bestätigen
- Timestamp auswählen: Kontextmenü und Popup öffnen, Timestamp auswählen und bestätigen/abbrechen
- GPT: Prompt bestätigen oder Brainstorm neu laden
- Raum-ID kopieren

Select-Elemente:

- Timestamp-Suche sortieren
- GPT-Brainstorms auswählen

Link-Elemente:

- Zwischen Registrierung und Login wechseln

Scroll-Elemente:

- Beigetretene Räume in scrollbarer Liste anzeigen
- Karten-Inhalt in scrollbarer Karte anzeigen
- GPT-Antworten in scrollbarem Textfeld anzeigen

Des Weiteren sind noch Div-Elemente zur Darstellung weiterer Information, die dem Nutzer angezeigt wird, verwendet.

4 Lösungsstrategie

Entwickler: NM, EP, NO

4.1 Technologieentscheidungen

4.1.1 Frontend

Autor(en): NM

Für das Entwickeln des Frontends haben wir uns entschieden, React in Kombination mit TypeScript zu wählen.

TypeScript ist eine typisierte Obermenge von JavaScript und bietet somit im Gegensatz zu diesem effektiveres Programmieren, bessere Codequalität und eine stabilere Anwendung.

React dagegen ermöglicht reaktive UIs, einen komponentenbasierten Code, eine nützliche Syntaxerweiterung (JSX) und im Gegensatz zu Angular ein schmales Boiler Plate. Zwar ist React kein komplettes Framework, ist aber durch externe Bibliotheken erweiterbar und ist so für nicht zu komplexe Anwendungen die bessere Wahl.

Des Weiteren haben wir Entwickler schon Erfahrung mit React in Kombination mit TypeScript, was ebenso Vorteile im Entwicklungsprozess und der fertigen Anwendung bietet.

4.1.2 Backend-Services

Autor(en): EP, NO

Wir haben TypeScript in Verbindung mit Express, GraphQL und Apollo verwendet, um den Gateway-Service zu entwickeln.

Express ist ein weit verbreitetes Routing-Tool, das uns ermöglicht hat, die Kommunikation zwischen dem Frontend und dem Gateway-Service zu realisieren.

Apollo ist ein effektives Werkzeug zur Implementierung von GraphQL-Servern und -Clients. Wir haben uns für Apollo entschieden, da wir GraphQL-Subscriptions nutzen möchten und Apollo eine einfache Integration dieser Funktionalität ermöglicht.

GraphQL ist eine leistungsfähige Abfragesprache für APIs. Wir haben uns für GraphQL entschieden, um eine API in unserem Microservice zu erstellen, die das Error-Handling vereinfacht und ein festes Schema bietet. Dadurch können wir auf der Client-Seite genau auswählen, welche Daten wir von der Abfrage benötigen.

Die Verwendung von GraphQL ermöglicht eine strukturierte API, die sowohl Sicherheit als auch Wartbarkeit gewährleistet.

In den Diensten Room-Service, Gpt-Service und Auth-Service haben wir TypeScript in Verbindung mit Express und GraphQL eingesetzt.

Den Apollo-Server haben wir in diesem Fall nicht integriert, da wir keine Subscriptions benötigten.

4.1.3 Datenspeicherung

Autor(en): NO

In unserer Anwendung verwenden wir je nach den spezifischen Anforderungen verschiedene Datenspeicherlösungen.

Für die Speicherung raumbezogener Daten, wie Rooms, Sections und Cards, haben wir uns für MongoDB entschieden. Das dokumentenbasierte Modell von MongoDB eignet sich gut für den Umgang mit verschachtelten Datenstrukturen. Wir verwenden zum Beispiel verschachtelte Strukturen wie Abschnitte und Karten innerhalb eines Raums. Dank der Flexibilität von MongoDB können wir komplexe Datenstrukturen problemlos speichern und abfragen. Darüber hinaus bietet MongoDB Skalierbarkeit und eine hohe Leistung, so dass es sich für die Verwaltung großer Mengen von Raumdaten eignet. Um die Daten klar strukturiert zu haben, wurden Schemen für die verschiedenen Collections verwendet

Für die Benutzerauthentifizierungsdaten setzen wir SQLite als Datenbanksystem ein. SQLite ist eine leichtgewichtige, dateibasierte Datenbank, die sich hervorragend für die Verwaltung strukturierter Daten mit geringer bis mittlerer Komplexität eignet. Wir haben uns für SQLite für die Benutzerauthentifizierung entschieden, weil es einfach und leicht einzurichten ist und in sich geschlossen ist und nur minimale Abhängigkeiten erfordert. Außerdem gewährleistet SQLite durch die Unterstützung von ACID-konformen Transaktionen die Integrität und Zuverlässigkeit der Daten.

Für die Interaktion mit der SQLite-Datenbank setzen wir Prisma ein, ein ORM-Tool (Object-Relational Mapping). Prisma vereinfacht die Interaktion durch die Bereitstellung einer intuitiven und typsicheren API. Es erzeugt einen Prisma-Client, der eine nahtlose Kommunikation mit der Datenbank unter Verwendung einer hochentwickelten Abfragesprache ermöglicht. Die Code-Generierungsfunktionen von Prisma machen manuelle SQL-Abfragen überflüssig, was die Produktivität der Entwickler erhöht und die Wahrscheinlichkeit von SQL-Fehlern verringert. Durch den Einsatz von Prisma profitieren wir von den Vorteilen eines ORM, wie z. B. der verbesserten Wartbarkeit des Codes und der erweiterten Datenbankabstraktion, während wir gleichzeitig SQLite als zugrunde liegende Datenspeichertechnologie nutzen.

4.1.4 Cloud-Deployment

Autor(en): NO

Für das Cloud-Deployment haben wir uns für Microsoft Azure als Plattform entschieden.

Um unsere Anwendung in der Cloud bereitzustellen, haben wir uns für einen virtuellen Computer mit einem Linux-Betriebssystem entschieden. Durch die Verwendung eines

virtuellen Computers können wir die erforderliche Umgebung nach unseren Anforderungen konfigurieren und anpassen. Linux wurde gewählt, da es eine weit verbreitete und robuste Plattform ist, die sich gut für den Betrieb von Anwendungen in der Cloud eignet.

Nachdem wir den virtuellen Computer eingerichtet haben, haben wir im Terminal Docker Compose ausgeführt. Indem wir Docker Compose verwenden, konnten wir unsere Anwendung und ihre Abhängigkeiten in Containern definieren und sie einfach bereitstellen.

Darüber hinaus haben wir den Port für das Frontend freigegeben, damit es über den Standard-HTTP-Port 80 von außen erreichbar ist. Durch die Freigabe dieses Ports können Benutzer von außerhalb auf unser Frontend zugreifen und die Funktionen unserer Anwendung nutzen. Auch der Port für das Gateway wurde freigegeben, um die Kommunikation zwischen Frontend und Gateway zu ermöglichen und die anderen Container darüber ansprechen zu können.

4.1.5 Service-Kommunikation

Autor(en): EP

Der Gateway-Service ist der “Mittelsmann”, welcher zwischen dem frontend und dem Gpt-, Auth-, und Room-Service steht. Das Frontend fragt an dem Gateway an und der Gateway handelt die die Kommunikation zwischen unseren Micro-Services.

Für die Kommunikation haben wir GraphQL verwendet. Die Gründe dafür wurden in 4.1.2 schon zu Grunde gelegt.

4.1.6 Code-Richtlinien

Autor(en): NM, EP

Um innerhalb des Projekts nach einheitliche Code-Richtlinien zu arbeiten, haben wir uns für ESLint als Werkzeug zur Überprüfung des Quellcodes entschieden.

Des weiteren haben wir Prettier verwendet, um die Struktur des Codes einheitlich zu gestalten.

4.2 Top-Level-Architektur-Entscheidung

Autor(en): NM

Gemäß den technischen Randbedingungen [2.1] wird die Anwendung als Microservice-Architektur entwickelt. Dabei wird jeder Service einzeln containerisiert und in der Cloud deployed.

Die Anwendung ist unterteilt in Frontend und Backend, wobei das Backend aus 5 separaten Services besteht.

4.2.1 Gateway-Service

Der Gateway-Service ist die Schnittstelle zwischen Frontend und restlichen Services. Es bietet die komplette Schnittstelle, die das Frontend für die volle Funktionalität braucht, an

und vereint die Funktionalität der restlichen Services. Außerdem ist er für die Authentifizierung des JWT-Tokens zuständig.

4.2.2 Authentication-Service

Der Authentication-Service ist für alle Authentifizierungsvorgänge zuständig. Er bietet die Schnittstelle zum Login und zur Registrierung eines Nutzers an und gibt den zur Authentifizierung genutzten JWT-Token zurück. Zur Datenspeicherung nutzt dieser eine im gleichen Container liegende SQLite-Datenbank.

4.2.3 Room-Service

Der Room-Service bietet alle Schnittstellen für die Logik eines Raumes an. Dazu gehört, Raumdaten zu manipulieren, auszugeben und zu speichern. Die Daten werden in einer MongoDB in einem eigenen Service gespeichert.

4.2.4 MongoDB-Service

Dieser Service ist eine MongoDB-Datenbank als eigener Service (DaaS) und es werden dort alle Daten zu einem Raum gespeichert

4.2.5 GPT-Service

Der GPT-Service ist verantwortlich für die Kommunikation mit der OpenAI API. Er bietet dann erforderliche Funktionen als Schnittstelle für den Gateway-Service an.

5 Bausteinsicht

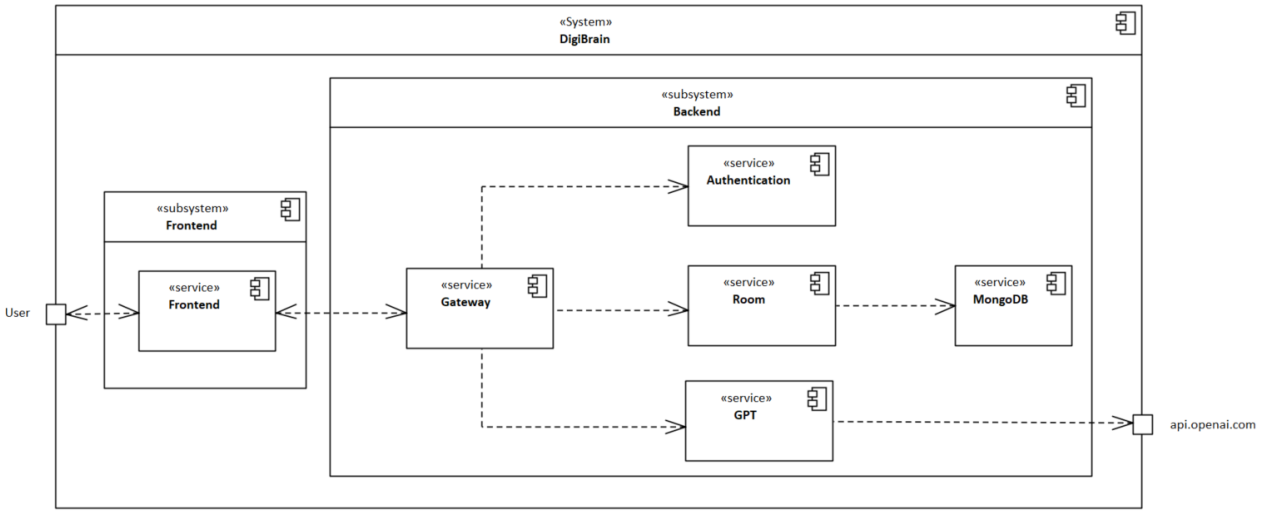
Im Folgenden wird das gesamte System und dessen Bestandteile mittels Black- und Whiteboxansichten aufgezeigt.

5.1 Ebene 1 (Whitebox Gesamtsystem)

Autor(en): NM, NO

Entwickler: NM, EP, NO

Das in der Kontextabgrenzung [3] dargestellte System DigiBrain wird in folgender Grafik als Whiteboxsystem dargestellt.



Das System wurde so zerlegt, um jeweils thematisch zusammenhängende Funktionen zu modularisieren. So bietet das der Gateway-Service eine fest definierte Schnittstelle für das Frontend, ohne dass dies sich um die Arch-itektur dahinter sorgen muss. Thematisch unabhängige Funktionalitäten sind so logisch voneinander getrennt und in sich geschlossen.

5.1.1 Frontend

5.1.1.1 Zweck/Verantwortung

Das Frontend stellt die Schnittstelle zwischen Anwendung und Nutzer dar. Es hat die Aufgabe, dem Nutzer eine Benutzeroberfläche bereitzustellen, über die er mit der Anwendung kommunizieren kann. Das Frontend ist somit für die Darstellung von Informationen, die Verarbeitung von Benutzerinteraktionen und ggf. die Kommunikation mit dem Gateway verantwortlich.

5.1.1.2 Schnittstelle

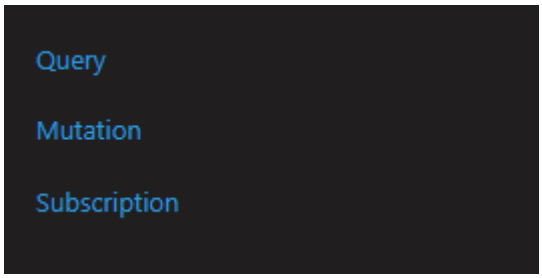
Die Schnittstelle des Frontends ist die Oberfläche. Die Funktionen, auf die der Nutzer Zugriff hat, sind in Kapitel 3.2.2 genauer geschildert.

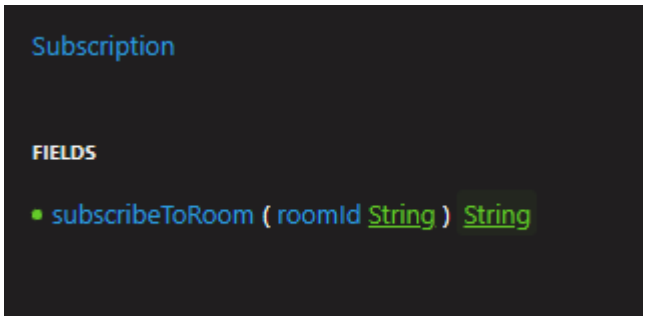
5.1.2 Gateway

5.1.2.1 Zweck/Verantwortung

Das Gateway ist für die Kommunikation zwischen den Services verantwortlich und delegiert die einzelnen Requests an die jeweiligen Microservices.

5.1.2.2 Schnittstelle



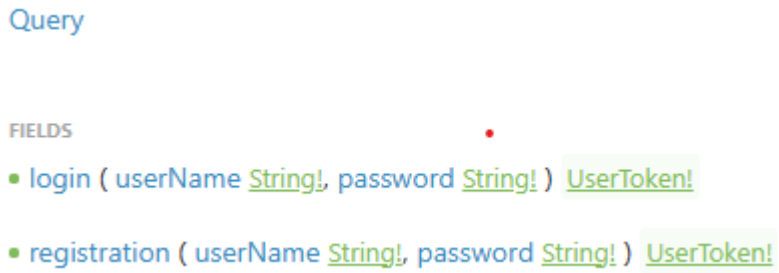


5.1.3 Authentication

5.1.3.1 Zweck/Verantwortung

Die Authentication ist für die Registrierung und den Login verantwortlich.

5.1.3.2 Schnittstelle



5.1.3.3 Ablageort/Datei(en)

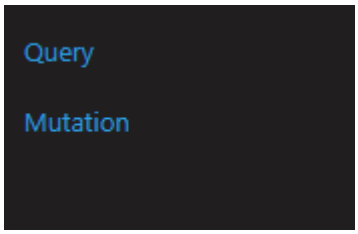
Gespeichert werden die User in einer SQLite Datenbank und danach zur MongoDB hinzugefügt.

5.1.4 Room

5.1.4.1 Zweck/Verantwortung

Der Room-Service übernimmt die Verwaltung der Räume. Über ihn kann man Räume erstellen, bearbeiten, beitreten und verlassen.

5.1.4.2 Schnittstelle



5.1.4.3 Ablageort/Datei(en)

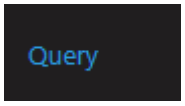
Gespeichert werden die Daten zu Rooms in einer MongoDB.

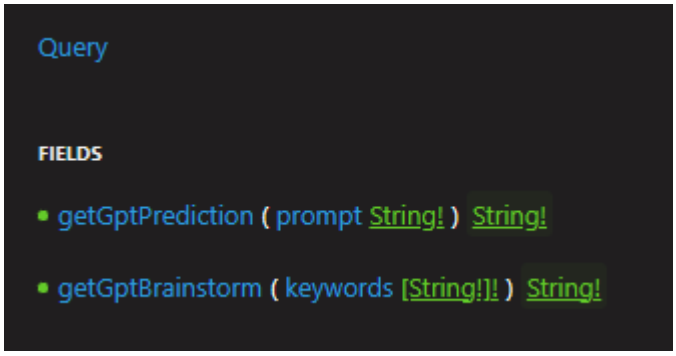
5.1.5 GPT

5.1.5.1 Zweck/Verantwortung

Der-GPT Service ist für die Kommunikation mit OpenAI verantwortlich.

5.1.5.2 Schnittstelle





5.1.5.3 Ablageort/Datei(en)

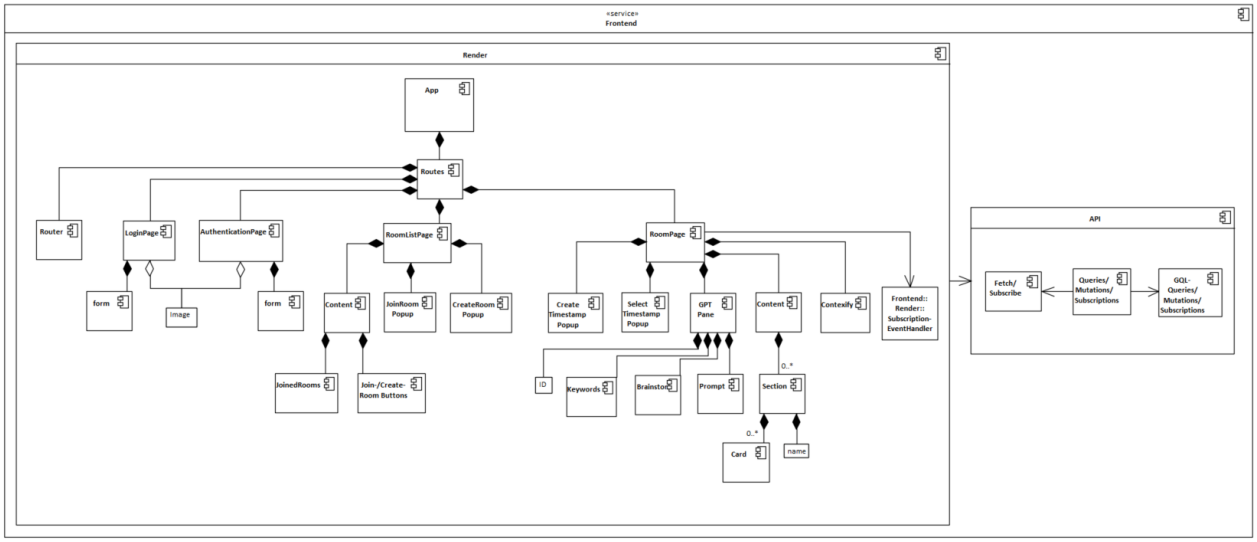
Die GPT Informationen zu den Keywords werden mit der zugehörigen RoomID in der MongoDB gespeichert.

5.2 Ebene 2

5.2.1 Whitebox Frontend

Autor(en): NM

Entwickler: NM



Das Frontend an sich besteht aus zwei Teilen. Zum einen die gerenderte App und zum anderen die API. Die gerenderte App kann dabei auf die API zugreifen, d.h. dass theoretisch jede Komponente im „Render“-Teil des gezeigten Diagramms auf die API zugreifen könnte.

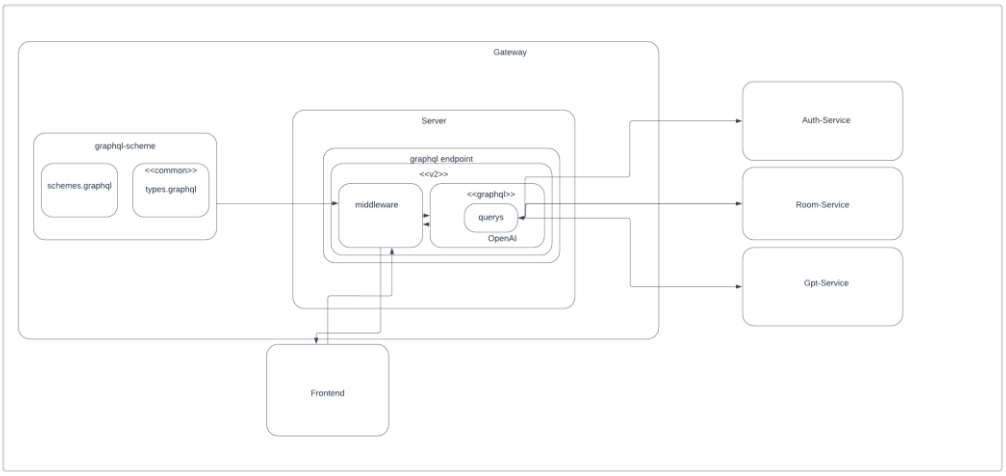
Die gerenderte App besteht aus vielen React-Komponenten. Diese React-Komponenten stellen jeweils eine UI-Komponente mit dazugehöriger Logik dar. Ganz oben ist die App. Diese hat die Routen der App als direkte Kinder. Jede Route ist ein URL-Pfad und rendert jeweils eine Page (Ausnahme „Router“: dient nur als Router für nicht vergeben Pfade). Die Pages sind sinnvoll voneinander getrennt und bieten so einen modularisierten Aufbau des Frontends.

Der API-Baustein besitzt zum einen eine Fetch-Funktion für http-Requests, als auch einen ApolloClient zum Herstellen einer WebSocket-Verbindung. Mithilfe der GraphQL-Queries bietet die API der gerenderten App dann fertige TypeScript-Funktionen zur Kommunikation mit dem Backend.

5.2.2 Whitebox Gateway

Autor(en): EP, NO

Entwickler: EP



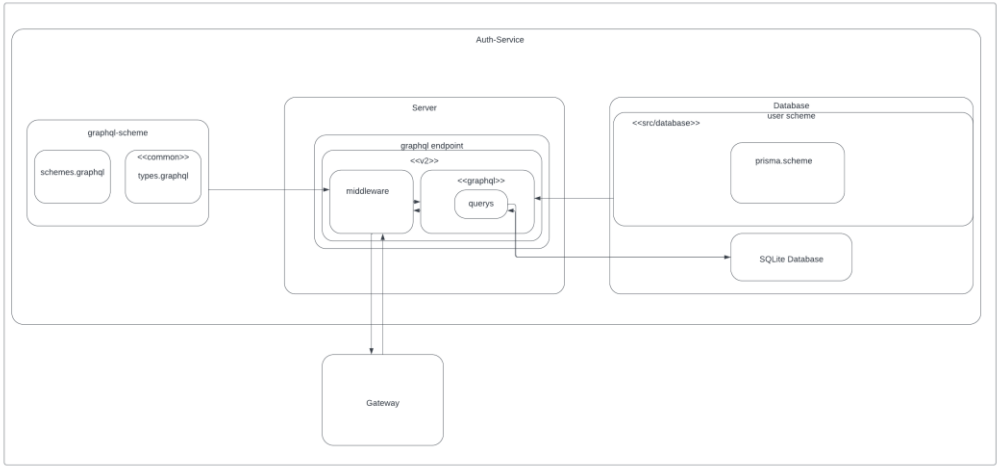
Das Gateway besteht aus der GraphQL middleware, welche die API zur Verfügung stellt. Über diese kann das Frontend mit dem Gateway kommunizieren.

Neben der API besitzt das Gateway noch die Graphql Schnittstelle zu den weiteren Micro-Services. Über diese kann das Gateway mit den anderen Services kommunizieren.

5.2.3 Whitebox Auth-Service

Autor(en): EP, NO

Entwickler: EP, NO



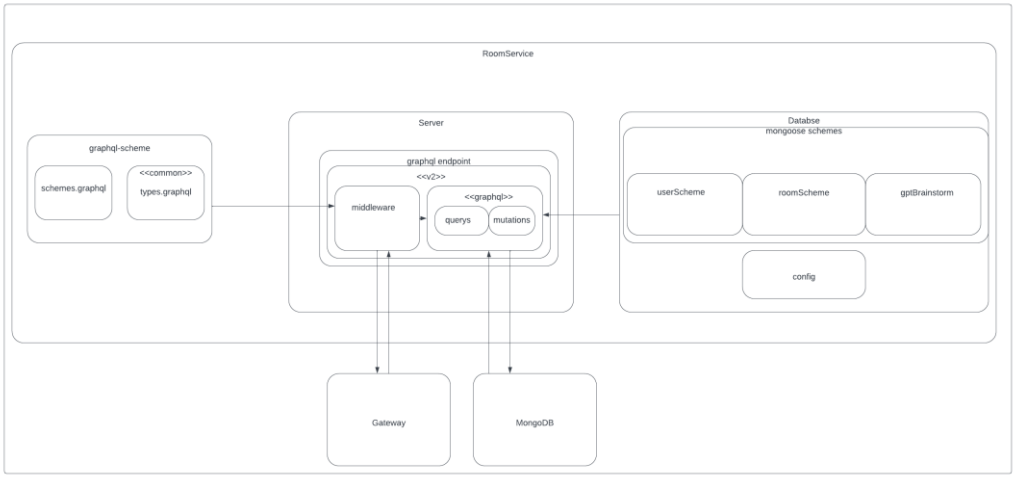
Der Auth-Service besteht aus der GraphQL Middleware, welche die API zur Verfügung stellt. Über diese kann der Gateway mit dem Auth-Service kommunizieren.

Im Auth-Service befindet sich außerdem noch die SQLite Datenbank in welcher die Anmeldedaten der Benutzer abgespeichert werden

5.2.4 Whitebox Room-Service

Autor(en): EP, NO

Entwickler: EP, NO



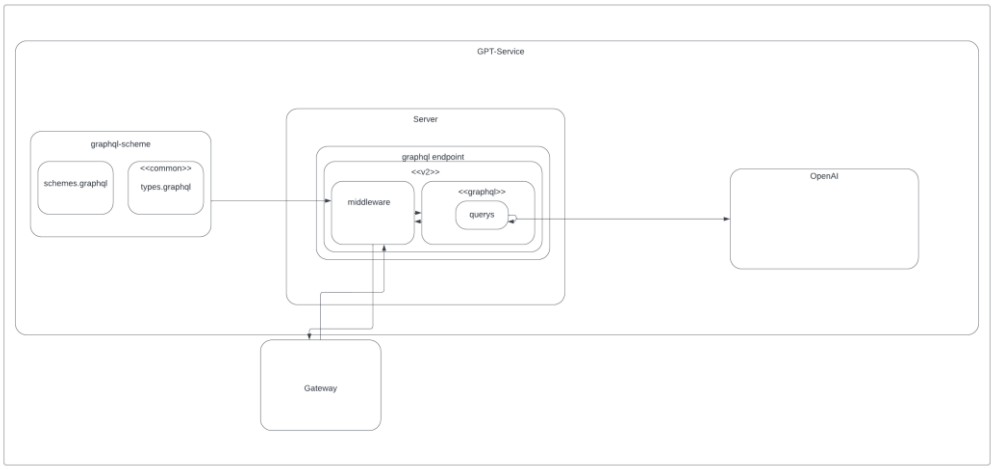
Der Room-Service besteht aus der GraphQL Middleware, welche die API zur Verfügung stellt. Über diese kann der Gateway mit dem Room -Service kommunizieren.

Über die Queries im Room-Service wird die MongoDB angesprochen.

5.2.5 Whitebox GPT-Service

Autor(en): EP, NO

Entwickler: EP



Der GPT-Service besteht aus der GraphQL Middleware, welche die API zur Verfügung stellt. Über diese kann der Gateway mit dem GPT -Service kommunizieren.

5.2.6 MongoDB-Service

Er Diagramm kommt in punkt 8.1

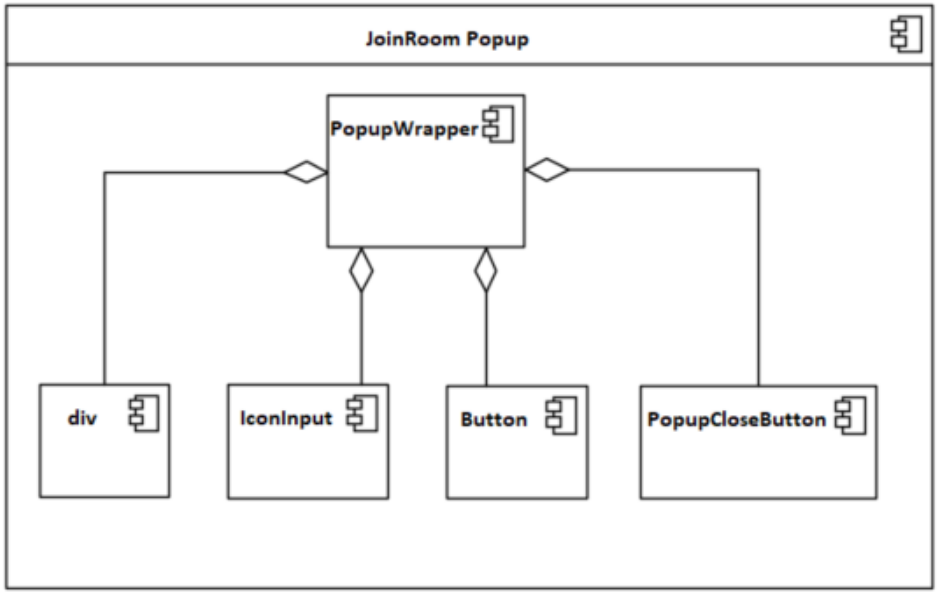
5.3 Ebene 3

5.3.1 Frontend

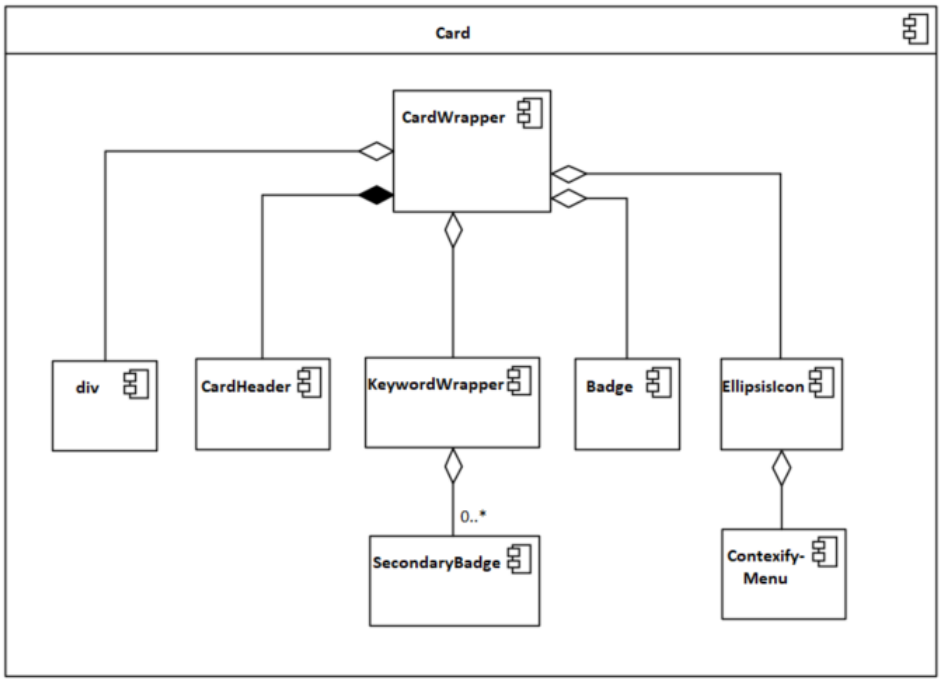
Autor(en): NM

Die nächsten beiden Diagramme zeigen zwei Beispiel-React-Komponenten, die in Abschnitt 5.2.1 als Blackbox dargestellt werden. Eine Aggregation zeigt an, dass diese Komponente auch von anderen Komponenten genutzt wird.

5.3.1.1 Whitebox Frontend.JoinRoomPopup



5.3.1.2 Whitebox Frontend.Card



6 Laufzeitsicht

Autor(en): NM

Im Folgenden sind die wichtigsten Laufzeitszenarien dokumentiert. Dabei können drei Grundszenarien beobachtet werden:

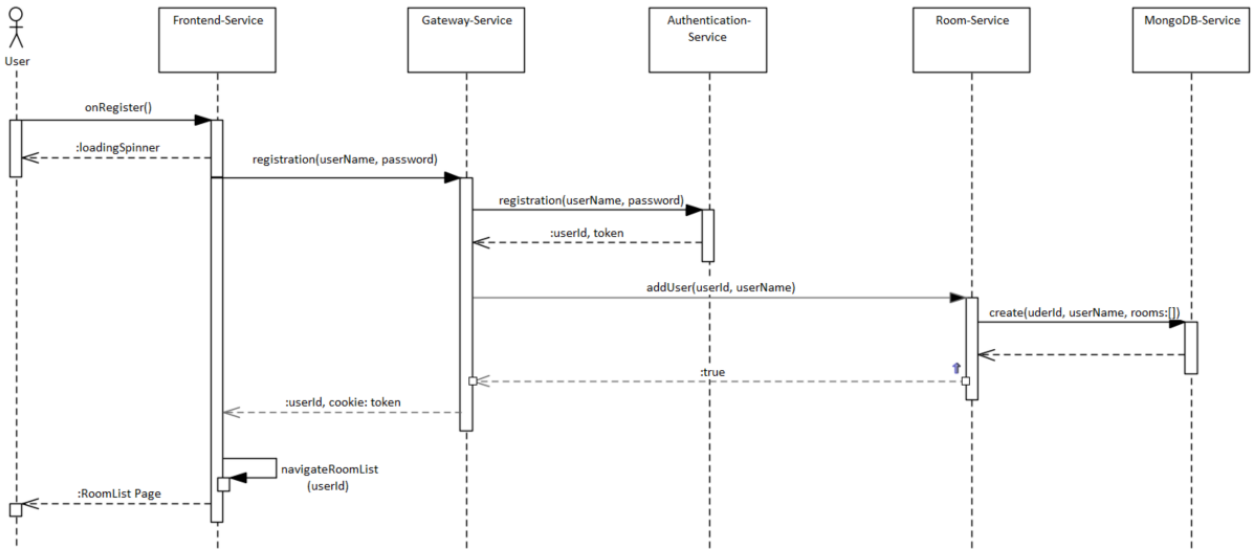
- Die Query: Daten werden vom Backend angefragt.

- Die Mutation: Daten werden verändert.
- Die Subscription: Das Backend sendet ein Event an das Frontend.

Nicht beschriebene Use-Cases können so nach Einordnung der drei Grundszenarien als ungefähre Variation betrachtet werden.

6.1 Nutzer registrieren (Query)

6.1.1 Erfolgsfall



Wenn ein Nutzer auf den „Register“-Button klickt, zeigt dieser einen Ladekreis an. Benutzername und Passwort werden dann über das Gateway and den Authentication-Service gesendet. Dieser legt den Nutzer in der SQLite Datenbank an und erstellt die Nutzer-ID, sowie den JWT-Token. Der JWT-Token und die Nutzer-ID werden an das Gateway gesendet, das daraufhin den Nutzer über den Room-Service in der MongoDB-Datenbank anlegt. Daraufhin gibt das Gateway die Nutzer-ID an das Frontend zurück und setzt den JWT-Token als Cookie. Mit der Nutzer-ID kann das Frontend dann zur RoomList-Page navigieren.

6.1.2 Fehlerfall

Wenn irgendein Service nicht das zurückgibt, was im Erfolgsfall definiert ist, bekommt der Nutzer eine Fehlermeldung

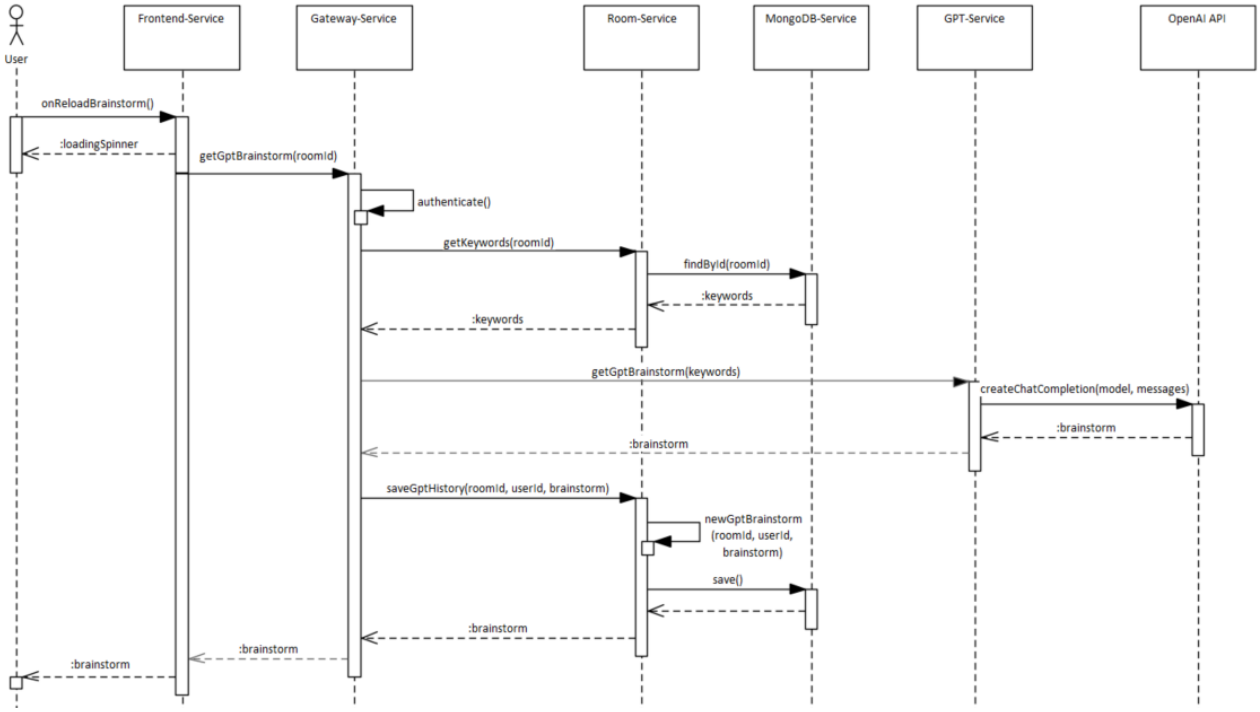
6.1.3 Grundszenario

Das Query-spezifische in diesem Laufzeitszenario ist der Ladekreis nach einer Aktion des Nutzers. Da die Serverantwort abgewartet werden muss, aber trotzdem eine Rückmeldung an den Nutzer gegeben werden sollte, wird dazu ein Ladekreis angezeigt.

Im Fehlerfall muss daher nicht eine lokale Veränderung rückgängig gemacht werden.

6.2 GPT Brainstorm (Query)

6.2.1 Erfolgsfall



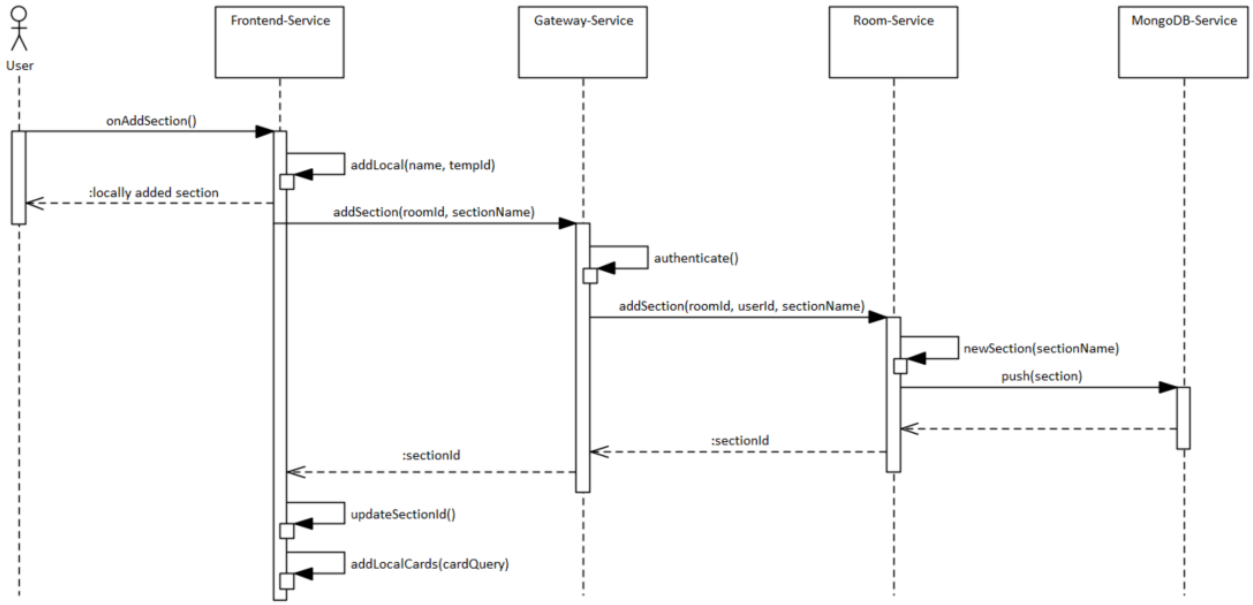
Wenn der Nutzer einen neuen GPT-Brainstorm anfragt, wird diesem direkt ein Ladekreis angezeigt. Das Frontend fragt dann beim Gateway einen neuen GPT-Brainstorm an. Dieses authentifiziert den Nutzer, fetched die Keywords vom Room-Service und gibt diese an den GPT-Service weiter. Der GPT-Service macht eine Request bei der OpenAI API und bekommt einen Brainstorm über die Keywords zurück. Diesen Brainstorm speichert das Gateway über den Room-Service in der MongoDB ab und gibt ihn an das Frontend zurück, welches das Ergebnis in einer Komponente dem Nutzer zur Verfügung stellt.

6.2.2 Fehlerfall

Wenn irgendein Service nicht das zurückgibt, was im Erfolgsfall definiert ist, bekommt der Nutzer eine Fehlermeldung.

6.3 Section hinzufügen (Mutation)

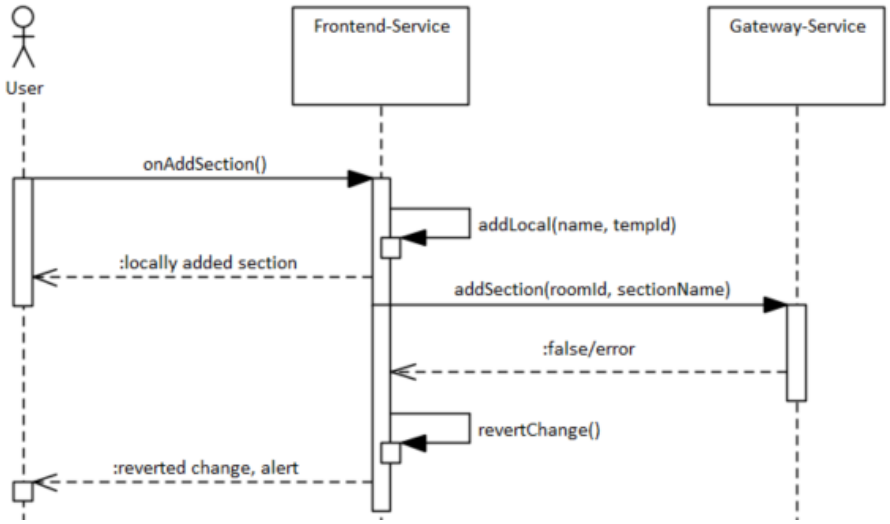
6.3.1 Erfolgsfall



Wenn der Nutzer eine Section hinzufügt, wird die Änderung direkt lokal im Frontend mit einer temporären ID durchgeführt und dem Nutzer angezeigt, um die Responsiveness zu gewähren. Daraufhin wird die Änderung über das Gateway gesendet. Dieses authentifiziert den Nutzer durch den Token in den Cookies. Daraufhin sendet das Gateway die Änderung an den Room-Service, der die Section mit ID erstellt und in die Datenbank hinzufügt. Die Section-ID wird dann über das Gateway an das Frontend gesendet. Im Frontend wird die temporäre ID durch die richtige ID aktualisiert. Karten, die während des Erwartens auf die richtige ID der Section hinzugefügt wurden, werden dann auch global hinzugefügt.

6.3.2 Fehlerfall

Folgendes Diagramm zeigt den Fehlerfall, wenn der Nutzer eine Section hinzufügt. Die Kommunikation der restlichen Services ist in diesem Diagramm nicht dargestellt, da es egal ist, wo der Fehler auftritt.



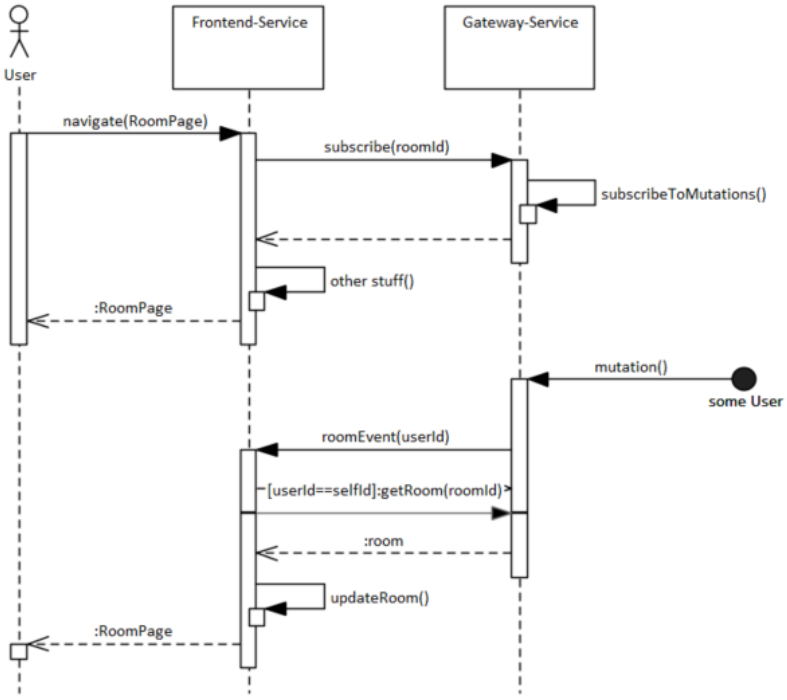
Wenn die Antwort des Servers einen Fehler oder False liefert, muss das Frontend die lokale hinzugefügte Section wieder entfernen machen. Dabei kann es nicht einfach den Zustand vor der Anfrage wiederherstellen, denn durch die nicht-blockierende Eigenschaft eines Raumes können jederzeit andere Änderungen gemacht, die dann durch eine wiederherstellende Funktion überschrieben werden würden. Deshalb muss die revertChange()-Funktion genau diese Änderung mittels ID rückgängig machen. Die revertChange()-Funktion muss aus diesem Grund für jede Mutation separat implementiert werden.

6.3.3 Grundszenario

Das Mutation-spezifische in diesem Laufzeitszenario ist die lokale Änderung des Raumes, obwohl die Serverantwort noch erwartet wird. Dies sichert die Responsiveness der Anwendung, erfordert aber auch eine komplexere Fehlerbehandlung als bei Query-Szenarien.

6.4 Zu Raum subscriben (Subscription)

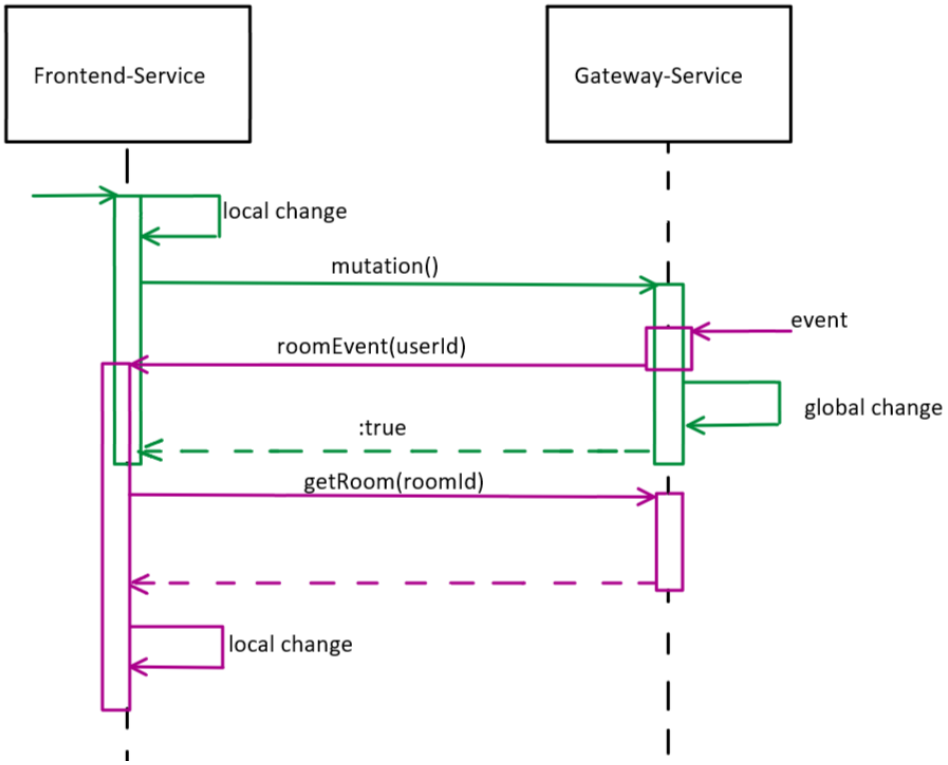
6.4.1 Erfolgsfall



Wenn die RoomPage geladen wird, wird auf ein Event auf diesem Raum subscribed. Sobald das Gateway angefragt wird, Daten eines Raumes zu verändern, wird ein Event mit der anfragenden Nutzer-ID an jeden subscribeten Nutzer gesendet. Dieser prüft, ob das Event von ihm selbst kommt, wenn nicht, fragt er den Raum beim Gateway an. Das Gateway gibt dann den geänderten Raum zurück (Service-Kommunikation bei getRoom() unvollständig, für dieses Diagramm aber unrelevant).

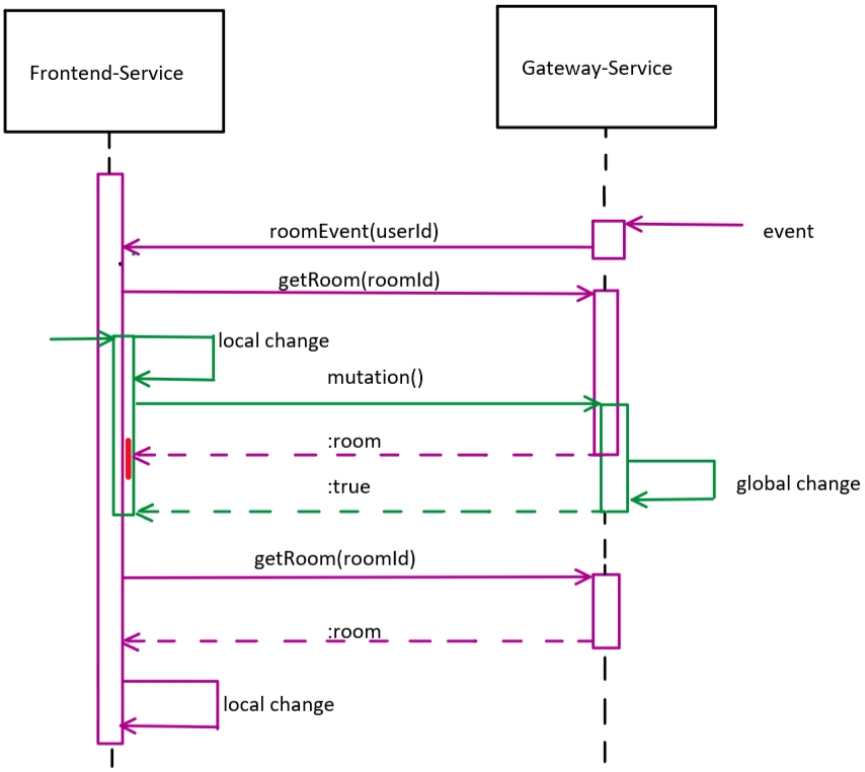
6.4.2 Sonderfall 1

Um den Sonderfall besser darzustellen, wird nur die Kommunikation zwischen Frontend und Gateway in vereinfachter Form und mit Farben veranschaulicht abgebildet.



Der Sonderfall tritt ein, wenn ein Subscription-Event ausgelöst wird (Lila), während der Client noch auf die Bestätigung einer lokalen Änderung (Grün) des Servers wartet. Das ausgelöste Event könnte noch den Status von vor der globalen Änderung abbilden und somit würde beim sofortigen Fetchen des Raumes die nur lokal angewendete Änderung überschrieben werden. Um dies zu verhindern, registriert der „Subscription Event Handler“ das eingetroffene Event und fetched den kompletten Raum erst dann, wenn alle lokalen Änderungen vom Server bestätigt wurden.

6.4.3 Sonderfall 2



Dieser Sonderfall tritt ein, wenn der Nutzer Änderungen durchführt (Grün), während gerade der Raum gefetched wird (Lila). Wird dieser Fall nicht behandelt, werden die lokalen Änderungen bei Eintreffen der Raum-Antwort überschrieben. Deshalb fängt der „Subscription Event Handler“ auch dieses Szenario ab. Kommt eine getRoom()-Response an, wird überprüft, ob gerade Antworten zu lokalen Änderungen erwartet werden. Ist dies der Fall, wird der gefetchedte Raum verworfen (Rot) und erst nach Bestätigung aller lokalen Änderungen wieder angefragt.

6.4.4 Fehlerfall

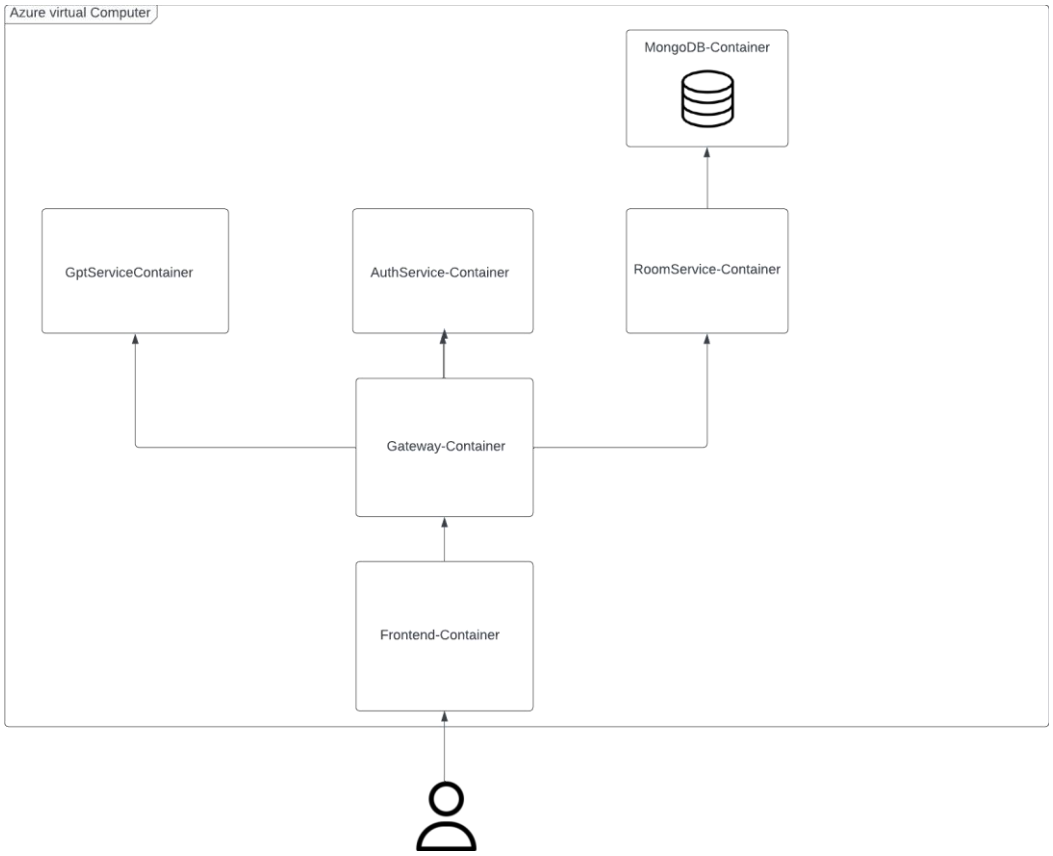
Wenn das Event nicht beim Frontend ankommt, entgeht die Änderung dem Nutzer und wird erst beim nächsten Event entdeckt.

Wenn die getRoom()-Funktion fehlschlägt, wird die Änderung verworfen, der Nutzer benachrichtigt und die Änderung beim darauffolgenden Event gefetched.

7 Verteilungssicht

Autor(en): NO

Entwickler: NO



7.1 Azure Virtual Computer

Die erste Ebene der digiBrain Web-Applikation ist der virtual computer auf der Azure Cloud, der die Applikation hostet.

Gewählt wurde dafür ein Computer mit Linux Betriebssystem (20.04) und 8GiB Ram, um die Applikation hosten zu können.

Der User ruft dafür den Virtuellen Computer mit dessen IP auf, um das Frontend auf dem Standart-HTTP Port (80) ansprechen zu können. Dafür wurde der Port freigegeben.

7.2 Docker Container

Jeder Microservice innerhalb des Virtuellen Computers und die MongoDB ist in seinem eigenen Docker Container, innerhalb eines Docker Networks.

Die Containerisierung in ein Docker Network fand mittels einer Docker Compose statt, um eine einfache Orchestrierung zu gewährleisten.

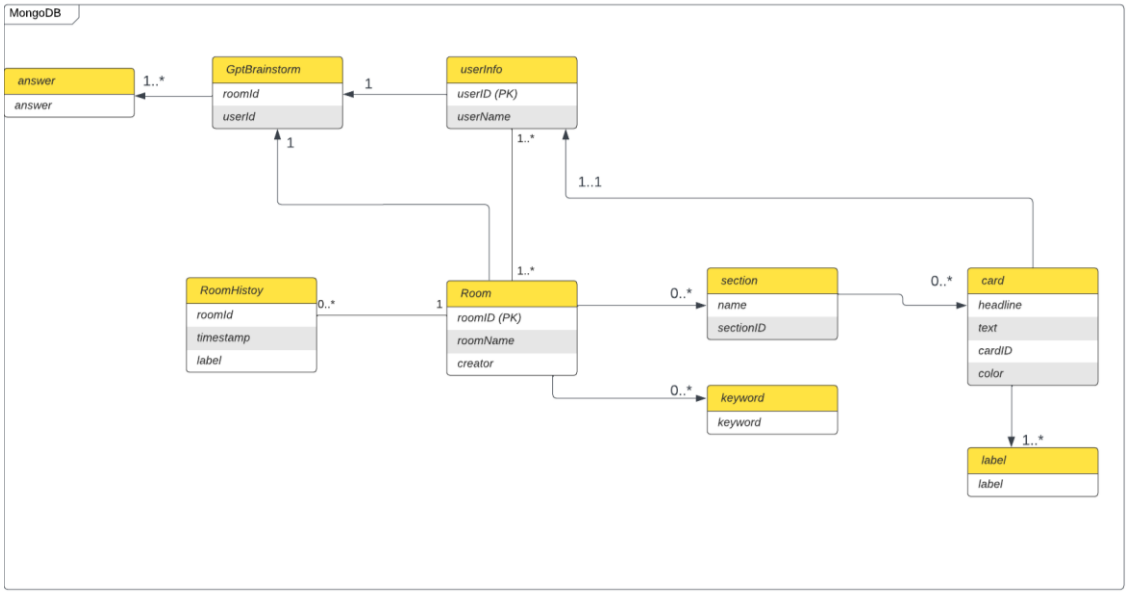
8 Querschnittliche Konzepte

Autor(en): NO

Nachdem im bisherigen Teil der Dokumentation nun genau auf den Aufbau der einzelnen Komponenten eingegangen wurde, werden im folgenden einzelne Komponenten genauer Betrachtet

8.1 *Datenmodell*

MongoDB:



Das genutzte Datenmodell zieht sich durch die Komplette Applikation, von MongoDB über Gateway, bis hin zum Frontend.

Das gezeigt ER-Diagramm zeigt die einzelnen Entitäten der MongoDB.

Im folgenden werden die Entitäten mit ihren Attributen genauer dargestellt, wie sie für die MongoDB gespeichert wurden.

User:

```
export type User = {
  userId: string;
  userName: string;
  rooms: Array<{ roomId: string }>;
} & Document;
```

Jeder User wird in der MongoDB mit dem userName, der UserId und den rooms, denen er beigetreten ist gespeichert.

Rooms:

```
export type Room = {
  id: ObjectId;
  name: string;
  creator: string;
  users: Array<string>;
  keywords: Array<string>;
  sections: Array<Section>;
} & Document;
export type Section = {
  id: ObjectId;
  name: string;
  cards: Array<Card>;
} & Document;
export type Card = {
  id: ObjectId;
  userId: string;
  userName: string;
  headline: string;
  color: string;
  text: string;
  labels: Array<string>;
} & Document;
```

Innerhalb jedes Rooms werden die id, name, der Ersteller (creator), die beigetretenen user, die Keywords gespeichert und die Sections, als Array of Sections. Ein Raum ist ein nested document, d.h. die Datenstruktur ist verschachtelt,

Innerhalb der Sections werden die id, der name und ein Array of cards gespeichert, wieder als nested document.

Die Cards speichern dann die id, die userId des Erstellers, damit nur der Ersteller die Karte editieren kann. Der userName, text, headline, color und die labels werden auch gespeichert und im Frontend angezeigt.

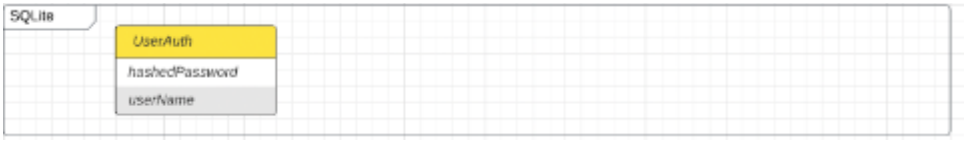
GptBrainstorm:

```
export type IGptBrainstorm = {
  roomId: string;
  userId: string;
  answer: Array<string>;
} & Document;
```

Zum GptBrainstorm werden die roomId, die userId und ein array of strings an answers gezeigt, damit jeder user die GPT-History zu dem jeweiligen Raum sehen kann.

SQLite:

Das folgende ER-Diagramm zeigt die einzelnen Entitäten der SQLite Datenbank(auch zu sehen unter 11.2 Informationsarchitektur)



Im folgenden wird die Entität nocheinmal genauer dargestellt.

```
generator client {
  provider = "prisma-client-js"
}

...
datasource db {
  provider = "sqlite"
  url      = "file:../../../auth-service/database/digiBrain.db"
}
...
model userAuth {
  userId   String  @id @default(cuid())
  password String
  userName String @unique
}
```

- Das Feld **userId** ist vom Typ **String** und wird als ID-Feld definiert (**@id**). Der **@default(cuid())**-Parameter gibt an, dass der Standardwert für das Feld automatisch generiert wird.
- Das Feld **password** ist vom Typ **String** und speichert das gehashte Passwort des Benutzers.
- Das Feld **userName** ist vom Typ **String** und wird als eindeutiges Feld (**@unique**) definiert, damit keine zwei benutzer den selben username haben können.

Der Codeausschnitt beschreibt die Konfiguration für den Prisma-Client und definiert das Datenmodell für die Entität "userAuth". Mit dem generierten Prisma-Client können dann Operationen wie das Erstellen, Lesen, Aktualisieren und Löschen von Daten in der Datenbank durchgeführt werden.

8.2 Subscription

GraphQL-Subscriptions sind ein Feature von GraphQL, das es ermöglicht, Echtzeitdaten von einem Server zu erhalten. Während traditionelle GraphQL-Abfragen (Queries) und Mutationen eine einmalige Antwort liefern, ermöglichen Subscriptions die kontinuierliche Aktualisierung von Daten, sobald sich etwas auf dem Server ändert.

Eine GraphQL-Subscription wird ähnlich wie eine GraphQL-Abfrage oder Mutation definiert, jedoch mit einem speziellen Schlüsselwort "subscription". Die Subscription

besteht aus einem Abonnementsnamen, optionalen Eingabevariablen und einer Abonnement-Payload, die festlegt, welche Daten der Client empfangen möchte.

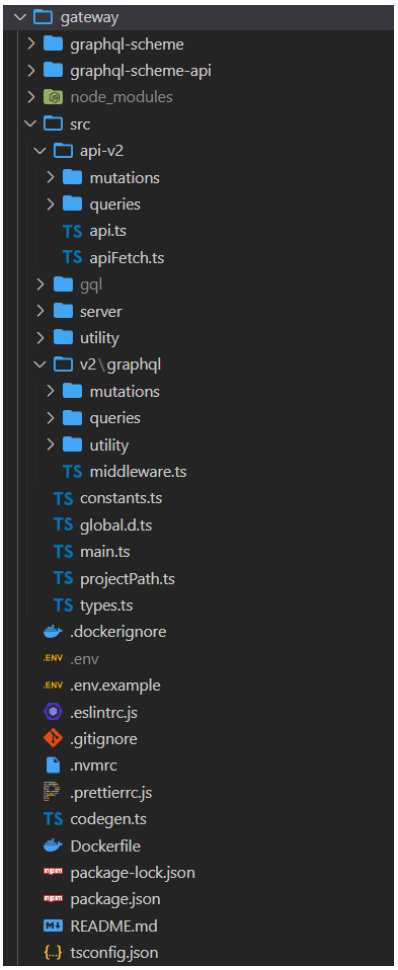
Einmal abonniert, bleibt die Verbindung zwischen dem Client und dem Server aktiv, und der Server sendet automatisch Aktualisierungen an den Client, sobald sich die relevanten Daten ändern. Dies ermöglicht Echtzeitkommunikation und sofortige Aktualisierungen, ohne dass der Client wiederholt Anfragen senden muss.

Typischerweise wird ein Websocket-Protokoll verwendet, um eine bidirektionale Kommunikation zwischen Client und Server herzustellen, die für GraphQL-Subscriptions erforderlich ist. Der Client öffnet eine Websocket-Verbindung zum Server und sendet seine Abonnementsanfrage. Der Server überwacht dann die relevanten Datenquellen und sendet die aktualisierten Daten über die Websocket-Verbindung an den Client. Der Client kann die empfangenen Daten entsprechend verarbeiten und darstellen.

GraphQL-Subscriptions sind besonders nützlich in Szenarien, in denen Echtzeitaktualisierungen erforderlich sind, z.B. in Chat-Anwendungen, Live-Feeds, Benachrichtigungssystemen oder Kollaborationswerkzeugen. Sie bieten eine effiziente Möglichkeit, Daten in Echtzeit zu übertragen, da nur die tatsächlich geänderten Daten an den Client gesendet werden, im Gegensatz zu Polling-Techniken, bei denen der Client wiederholt Anfragen sendet, um zu überprüfen, ob sich Daten geändert haben.

GraphQL-Subscriptions sind ein leistungsstarkes Feature, das die Flexibilität und Effizienz von GraphQL erweitert und es Entwicklern ermöglicht, Echtzeitfunktionen in ihre Anwendungen zu integrieren.

8.3 Microservice Aufbau



Hier wird de Aufbau des Gateways gezeigt. Die anderen Mico-Serivces sind ähnlich aufgebaut, bei ihnen fehlt jedoch der api-v2 Part.

Auf der ersten Ebene haben wir Standard-Files wie die package.json, die README.md, die .gitignore und die tsconfig.json.

Die .env Datei enthält wichtige Enviroment-Variablen, welche nicht veröffentlicht werden sollten wie zum Beispiel ein API-Key für OpenAI oder der Secret-Key für die JWTs.

Die .nvmrc enthält die Node version, welche für diesen Service verwendet wurde

Die .eslintrc.js enthält den die Configuration von ESLint.

Im graphql-scheme ordner befindet sich das GraphQL Schema, welches alle Queries für die API im Gateway enthalten.

Der graphql-scheme-api Ordner hingegen enthält alle Queries, welche für die Kommunikation mit den anderen Micro-Services enthalten.

Im src-Ordner befindet sich der Code, welcher den Server zum laufen bringt.

Es gibt den v2/graphql Pfad, in welchem die Queries und Mutations für die Gateway API stehen. In ihnen werden die verschiedenen Micro-Services angesprochen. Um diese überhaupt ansprechen zu können benötigt man den api-v2 Ordner. In diesem Ordner werden die Queries und Mutations für die Micro-Services angegeben.

Das bedeutet, es wird letztendlich nur eine Query aufgerufen, welche die Anfrage weiterleitet und die Antwort der weitergeleiteten Anfrage wieder zurückgibt.

Der server Ordner enthält den httpServer und die express Funktionen.

Die utility Ordner enthalten nützliche Funktionen wie bestimmte Handler oder validation Funktionen.

9 Architekturentscheidungen

Autor(en): EP

Verwendung von Docker für die Containerisierung: Entscheidung: Die Anwendung wird in Docker-Containern bereitgestellt. Begründung: Docker ermöglicht die Isolierung von Anwendungen und deren Abhängigkeiten in Containern. Dies erleichtert die Bereitstellung, Skalierung und Wartung der Anwendung, da sie unabhängig von der zugrunde liegenden Infrastruktur ausgeführt werden kann.

Einsatz von GraphQL für die API-Gestaltung: Entscheidung: Die Anwendung nutzt GraphQL als API-Schnittstellentechnologie. Begründung: GraphQL ermöglicht flexible und effiziente Datenabfragen, da Clients nur die Daten erhalten, die sie tatsächlich benötigen. Es vereinfacht die Entwicklung, da die API-Struktur unabhängig von den Client-Anforderungen ist und ermöglicht eine schnelle Iteration und Entwicklung von Frontend-Funktionen.

Verwendung von TypeScript als Programmiersprache: Entscheidung: Die Anwendung wird mit TypeScript entwickelt. Begründung: TypeScript ist eine typisierte Superset-Sprache von JavaScript, die statische Typisierung ermöglicht. Dadurch werden potenzielle Fehler frühzeitig erkannt und die Codequalität verbessert. TypeScript bietet auch bessere Entwicklungsunterstützung durch IDEs und verbesserte Dokumentation.

Verwendung von React.js für die Benutzeroberfläche: Entscheidung: Die Benutzeroberfläche der Anwendung wird mit React.js entwickelt. Begründung: React.js ist eine beliebte JavaScript-Bibliothek zur Entwicklung von Benutzeroberflächen. Es bietet eine komponentenbasierte Architektur, wodurch die Wiederverwendbarkeit von Code und die Effizienz bei der Entwicklung verbessert werden. React.js ermöglicht auch eine effiziente Aktualisierung der Benutzeroberfläche durch den Einsatz von virtuellem DOM.

Verwendung von Express als Webframework: Entscheidung: Die Anwendung verwendet das Express-Framework für die Entwicklung des Backends. Begründung: Express ist ein einfaches und flexibles Node.js-Framework für die Entwicklung von Webanwendungen. Es bietet eine robuste Routing-Engine, Middleware-Unterstützung und eine aktive Entwicklergemeinschaft. Express ermöglicht eine schnelle Entwicklung von RESTful APIs und die Integration von Middleware-Funktionen.

Einsatz von Apollo als GraphQL-Client: Entscheidung: Der Apollo-Client wird für die Kommunikation mit GraphQL-Servern verwendet. Begründung: Apollo bietet einen leistungsfähigen und erweiterbaren GraphQL-Client, der nahtlos mit React.js integriert werden kann. Es vereinfacht die Datenverwaltung, Abfrage-Caching und Fehlerbehandlung. Apollo bietet auch eine gute Entwicklererfahrung mit Tools wie Apollo DevTools und GraphQL Code Generator.

Microservices-Architektur: Entscheidung: Die Anwendung wird als Microservices-Architektur entworfen und entwickelt. Begründung: Eine Microservices-Architektur ermöglicht eine modulare Entwicklung und Skalierung der Anwendung. Sie fördert die Unabhängigkeit und lose Kopplung der einzelnen Dienste, wodurch sie leichter zu entwickeln, testen und warten sind. Jeder Microservice kann unabhängig bereitgestellt und skaliert werden, was die Flexibilität und Skalierbarkeit der Anwendung verbessert. Zudem ermöglicht es die Nutzung verschiedener Technologien und Datenbanken, um die spezifischen Anforderungen jedes Microservices optimal zu erfüllen.

Verwendung eines API-Gateways: Entscheidung: Die Anwendung verwendet ein API-Gateway zur Aggregation und Weiterleitung von Anfragen an die entsprechenden Microservices. Begründung: Ein API-Gateway fungiert als zentrale Schnittstelle für den Zugriff auf die verschiedenen Microservices. Es ermöglicht die Aggregation von Daten aus verschiedenen Diensten, die Durchführung von Authentifizierung und Autorisierung, Caching und andere Funktionen wie Lastausgleich und Fehlerbehandlung. Ein API-Gateway vereinfacht die Kommunikation zwischen den Clients und den Microservices und verbessert die Sicherheit und Leistung der Anwendung.

Auswahl der Datenbanken: Entscheidung: Die Anwendung verwendet sowohl SQLite als auch MongoDB als Datenbanken, abhängig von den Anforderungen der einzelnen Microservices. Begründung: SQLite ist eine leichte relationale Datenbank, die gut für lokale Entwicklungsumgebungen und kleine Anwendungen geeignet ist. MongoDB hingegen ist eine dokumentenorientierte NoSQL-Datenbank, die Skalierbarkeit und Flexibilität bietet. Die Wahl der Datenbank hängt von den spezifischen Anforderungen jedes Microservices ab. Durch die Verwendung verschiedener Datenbanken können wir die Vorteile beider Technologien nutzen und die Datenbanken an die spezifischen Anwendungsfälle anpassen.

Diese Architekturentscheidungen bieten eine Grundlage für den Entwurf und die Entwicklung der Anwendung. Sie wurden getroffen, um die Skalierbarkeit, Flexibilität, Wartbarkeit und Leistung der Anwendung zu verbessern und die Anforderungen der Stakeholder bestmöglich zu erfüllen. Es ist wichtig, diese Entscheidungen während des gesamten Entwicklungsprozesses zu berücksichtigen und gegebenenfalls zu überprüfen, um sicherzustellen, dass sie weiterhin den Zielen des Projekts entsprechen.

10 Risiken und technische Schulden

Autor(en): EP

Übermäßige Netzwerklatenz: GraphQL-Abfragen können komplexe und umfangreiche Datenanforderungen enthalten. Wenn die Netzwerklatenz hoch ist, kann dies zu langen Wartezeiten führen, insbesondere wenn mehrere GraphQL-Abfragen nacheinander gesendet werden. Maßnahme: Optimieren Sie die Netzwerkkommunikation durch Batching von Abfragen, Reduzierung der Datenmenge oder Caching von Abfrageergebnissen.

Skalierbarkeitsprobleme: Wenn Ihre Anwendung stark frequentiert ist und viele gleichzeitige Benutzeranfragen verarbeiten muss, können Engpässe in der Serverleistung auftreten. Dies kann zu langsamen Antwortzeiten oder sogar zu Ausfällen führen. Maßnahme: Skalieren Sie Ihre Serverinfrastruktur horizontal, verwenden Sie Caching-Strategien und implementieren Sie geeignete Lastverteilungsmechanismen.

Sicherheitsrisiken: Unsachgemäße Konfiguration oder Implementierung von Sicherheitsmechanismen kann zu Sicherheitslücken führen, wie beispielsweise unzureichende Validierung von Eingaben, unzureichende Berechtigungsprüfungen oder Exposition sensibler Daten. Maßnahme: Implementieren Sie bewährte Sicherheitspraktiken, wie z.B. Eingabevalidierung, Berechtigungsprüfungen, sichere Authentifizierung und Verschlüsselung.

Komplexe Datenbeziehungen: GraphQL ermöglicht komplexe Datenabfragen, die über verschiedene Datenquellen und Beziehungen hinweggehen können. Dies kann zu Schwierigkeiten bei der Modellierung und Verwaltung von Daten führen, insbesondere wenn die Datenquellen heterogen sind. Maßnahme: Sorgfältige Planung und Gestaltung der Datenmodellierung, Implementierung effizienter Resolver-Funktionen und Verwendung von Caching-Strategien, um Datenzugriffe zu optimieren.

Mangelnde Dokumentation: Da GraphQL flexible Datenstrukturen ermöglicht, kann es schwierig sein, die Schnittstellen und ihre Anforderungen zu verstehen, insbesondere wenn die Dokumentation unzureichend ist. Maßnahme: Erstellen Sie umfassende Dokumentation für Ihre GraphQL-Schnittstellen, einschließlich der verfügbaren Abfragen, Mutationen, Subscriptions und ihrer Struktur.

Komplexe Build-Konfiguration: Die Kombination von React.js, Node.js, TypeScript, Express und OpenAI kann eine komplexe Build-Konfiguration erfordern, insbesondere wenn sie in einer einzigen Anwendung zusammengeführt werden. Dies kann zu Konfigurationsproblemen, Inkompatibilitäten und schwieriger Wartbarkeit führen. Maßnahme: Verwenden Sie geeignete Build-Tools, Automatisierung und Best Practices für die Konfiguration, um eine reibungslose Integration und Wartbarkeit sicherzustellen.

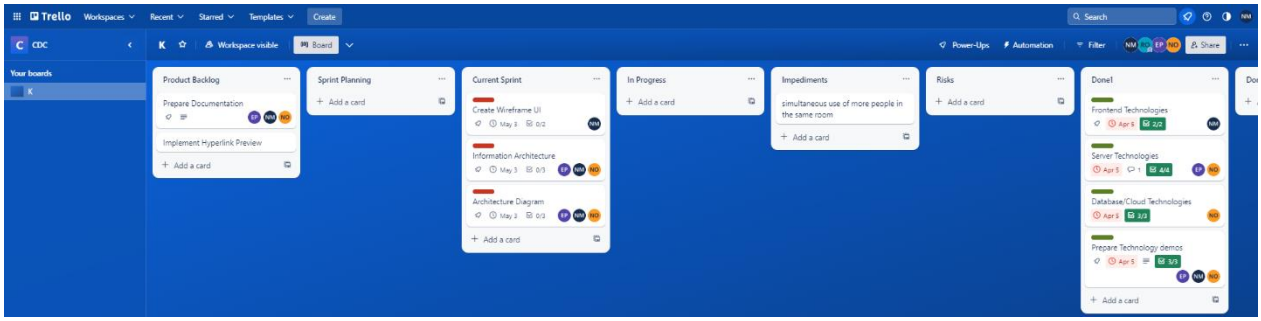
11 Zusatz

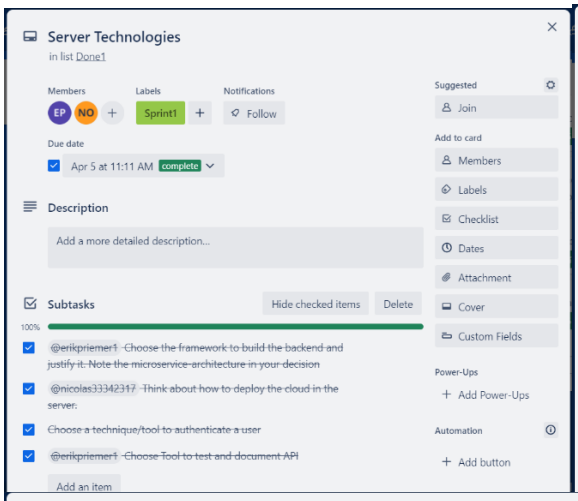
Author: NM

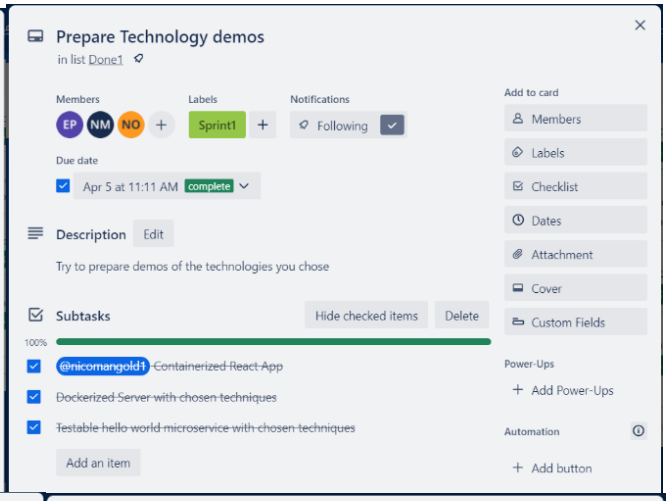
11.1 Projektlog

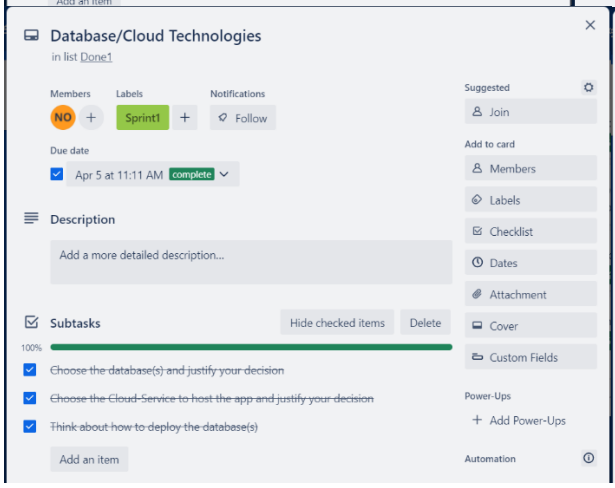
11.1.1 Exploratory Sprint Review

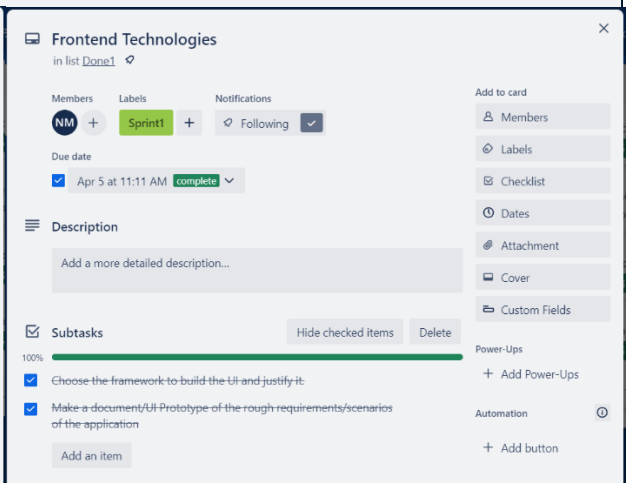
Übersicht





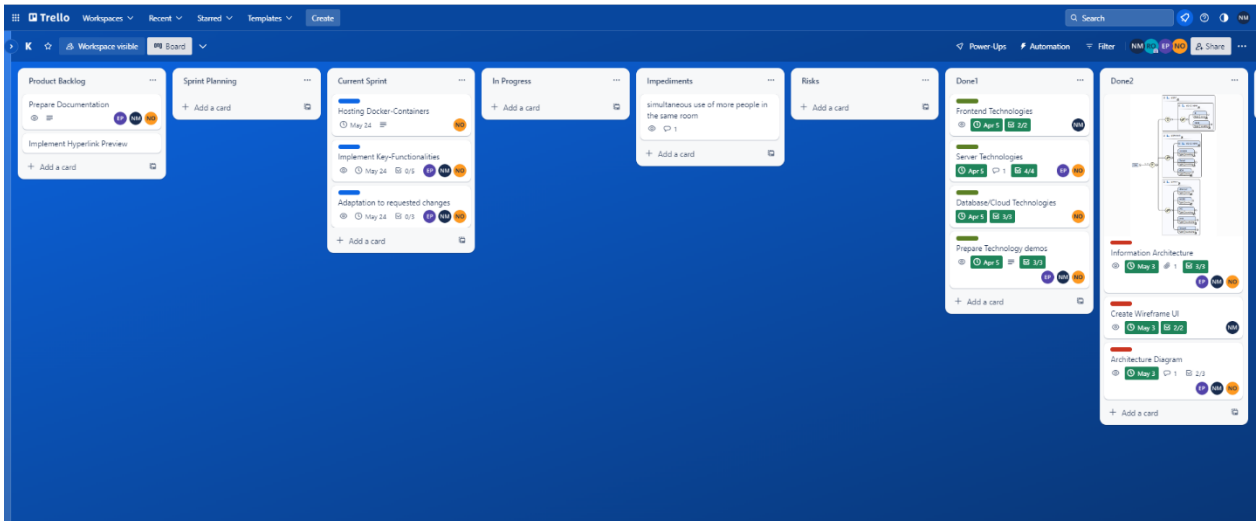


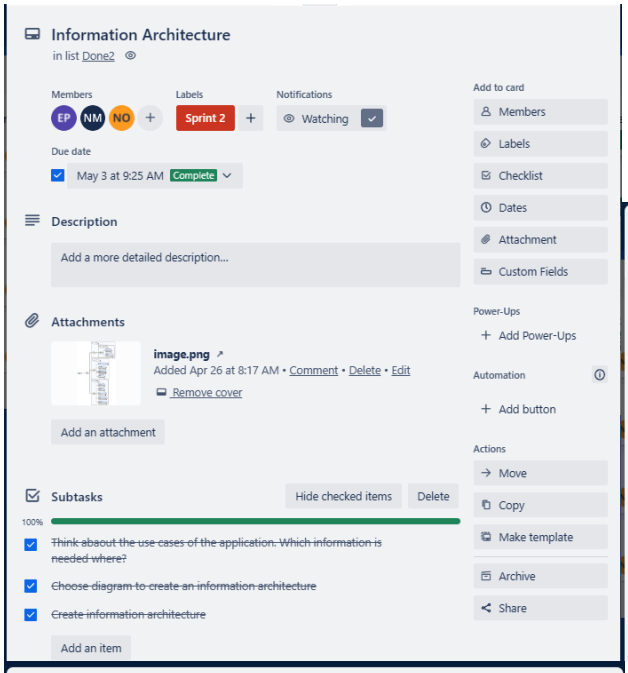


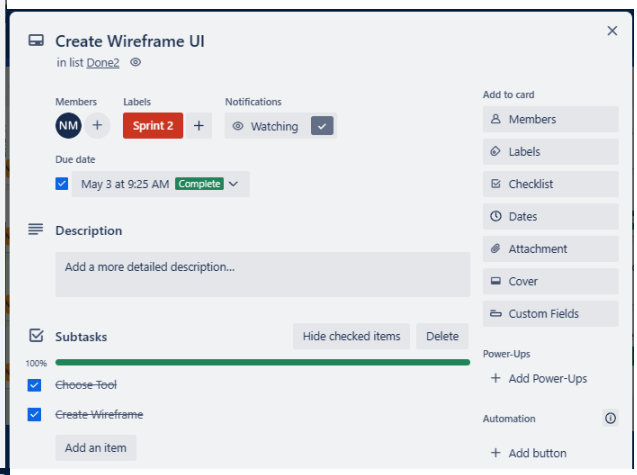


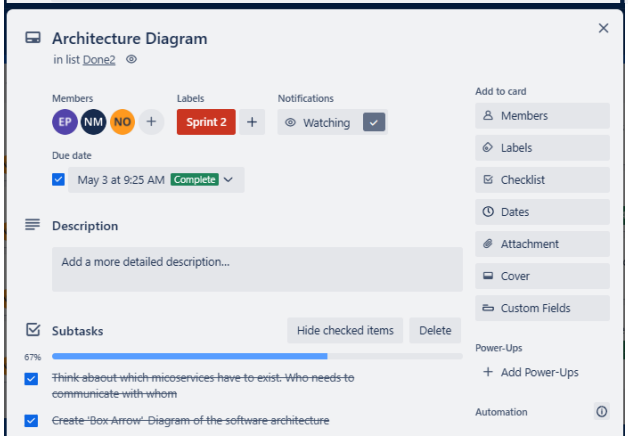
11.1.2 Architectural Spike Review

Übersicht:



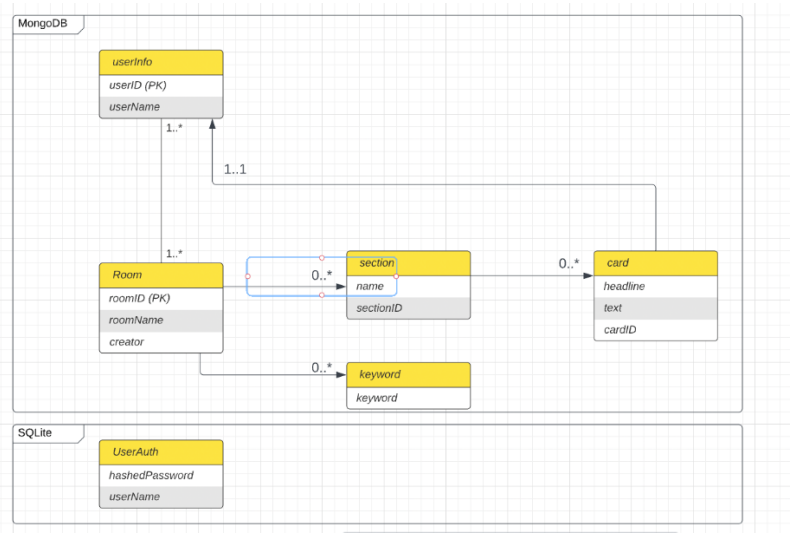




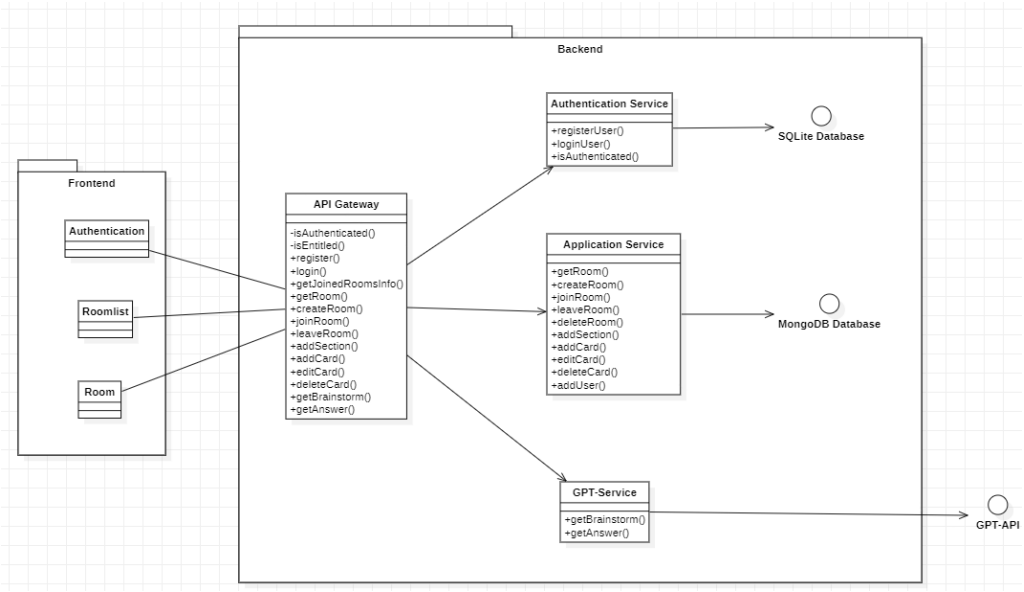


Informationsarchitektur (EP, NO):


```
{
 "users": [
  {
  "userId": "string",
  "userName": "string",
  "rooms": [
   "string"
  ]
  }
 ],
 "rooms": [
  {
   "roomId": "string",
   "roomName": "string",
   "creator": "string",
   "users": [
    "string"
   ],
   "keywords": [
    "string"
   ],
   "sections": [
    {
     "sectionId": "string",
     "name": "string",
     "cards": [
      {
       "cardId": "string",
       "userId": "string",
       "headline": "string",
       "text": "string"
      }
     ]
    }
   ]
  }
 ]
}
```

Architekturdiagramm (NM):

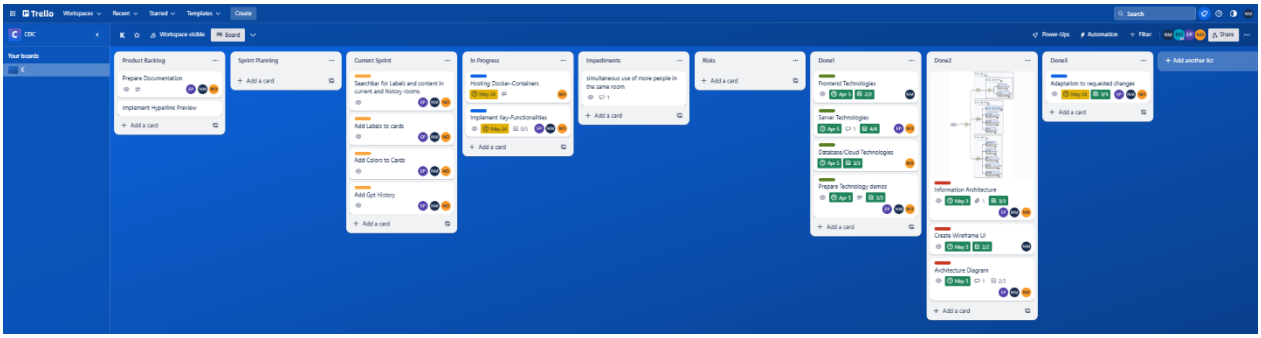


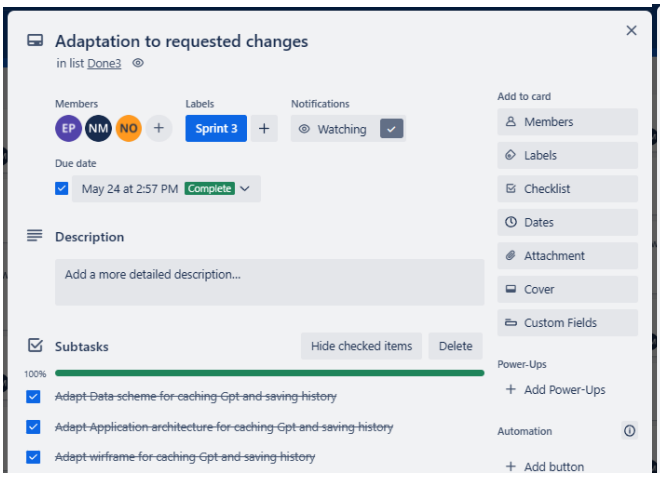
Wireframe (NM):

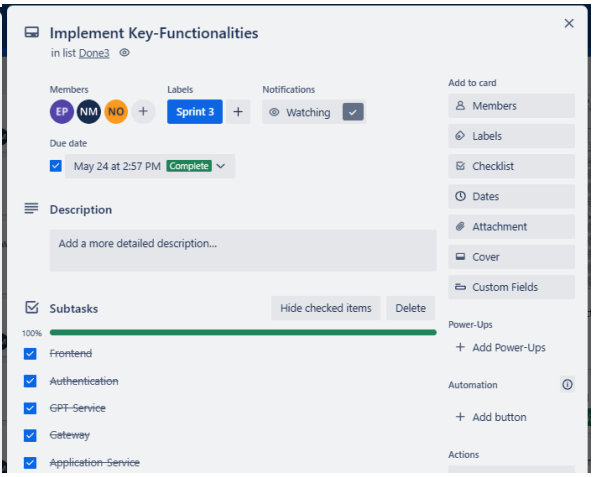
wireframe.pdf

11.1.3 Alpha Sprint Review

Übersicht

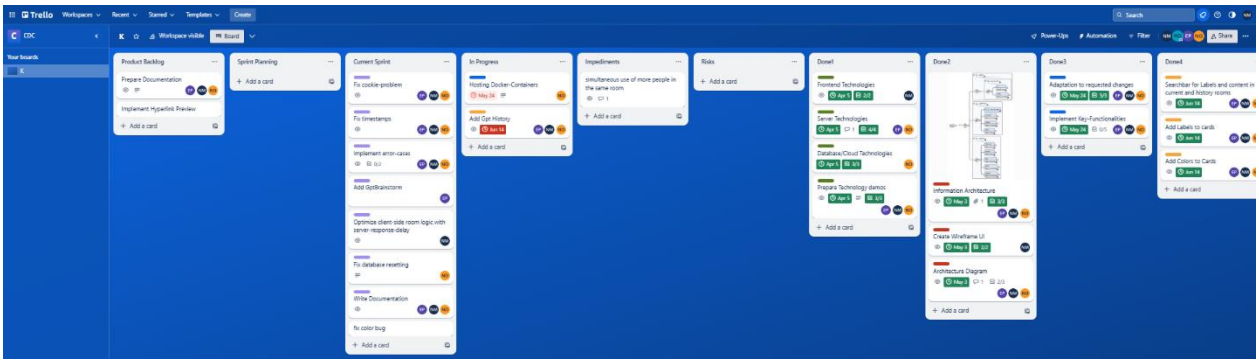






11.1.4 Beta Sprint Review

Übersicht



11.1.5 Quality Sprint Review

Übersicht