

# Google Docs Light

Workshop Web - FS22

23. Mai 2022

Studenten P. Schmucki, J. Villing, K. Zellweger

Dozenten D. König, S. Meichtry, J. Luthiger

Studiengang Informatik

Hochschule Hochschule für Technik

## Inhaltsverzeichnis

<b>1</b>	<b>Summary</b>	<b>1</b>
<b>2</b>	<b>Technologie Stack</b>	<b>2</b>
2.1	Lösungsstrategie . . . . .	2
2.2	Backend Server . . . . .	2
2.3	Frontend Clients . . . . .	2
2.4	Datenbank System . . . . .	2
<b>3</b>	<b>Systemübersicht</b>	<b>4</b>
3.1	Services . . . . .	4
3.2	Sequenz . . . . .	4
3.3	Applikationsprotokoll . . . . .	4
3.4	Benutzerverwaltung . . . . .	4
<b>4</b>	<b>Frontend</b>	<b>5</b>
4.1	Aufbau . . . . .	5
4.2	State- und Konfliktmanagment . . . . .	9
4.3	Fehler Behandlung . . . . .	11
<b>5</b>	<b>Backend</b>	<b>13</b>
5.1	Aufbau . . . . .	13
5.2	API . . . . .	14
5.3	Komponenten . . . . .	15
5.4	Abläufe . . . . .	19
5.5	Zustands- und Konfliktmanagment . . . . .	21
<b>6</b>	<b>Testing</b>	<b>23</b>
6.1	Frontend . . . . .	23
6.2	Backend . . . . .	23
6.3	End to End Test . . . . .	23
<b>7</b>	<b>Deployoment</b>	<b>24</b>
<b>8</b>	<b>Ausblick</b>	<b>25</b>
<b>9</b>	<b>Fazit</b>	<b>25</b>
	<b>Abbildungsverzeichnis</b>	<b>26</b>

## 1 Summary

## 2 Technologie Stack

Eine zentrale Anforderung an das System ist die konsistente und verzögerungsfreie Darstellung eines Dokuments auf mehreren Klienten. Die Wahl eines geeigneten Kommunikationsprotokolls ist die Grundlage für eine erfolgreiche Lösung.

Wir verwenden HTTP-Event Streams als Grundlage für die Kommunikation zwischen dem Backend Server und den Klienten. Als konkrete Implementation dieser Technologie setzen wir Spring-WebFlux ein. Die weitere Technologieauswahl orientiert sich an diesem Grundsatz Entscheid.

### 2.1 Lösungsstrategie

Um die verteilte Dokumentenbearbeitung zu ermöglichen, verwenden wir eine Variante des Command Patterns als Lösungsansatz. Dabei sollen die eingehenden Änderungen am Dokument als einzelne Commands modelliert werden. Die eingehenden Commands werden vom Backend verarbeitet und an alle Teilnehmer weitergeleitet. Konflikte werden dabei auf dem Server gelöst. Wenn nötig werden dabei zusätzliche Commands generiert, welche die Konflikte auflösen.

### 2.2 Backend Server

Spring WebFlux ist integriert in das Spring Boot Ökosystem und benötigt daher eine zugrundeliegende JVM. Sprachen die auf der JVM aufbauen, haben den Vorteil, dass sie System Interoperabel sind.

Anstatt Java setzen wir jedoch auf Kotlin als Backend Sprache. Bis jetzt hat kein Mitglied des Projektteams nennenswerte Erfahrung mit Kotlin und wir möchten diese Gelegenheit nutzen, die Sprache in einem Projekt näher kennenzulernen. Wir erwarten die nachfolgenden Vorteile:

- Robuste Implementierung dank Null Safety
- Weniger Boilerplate und damit übersichtlichere Implementierung
- Effiziente und Übersichtliche Anwendung von Streams
- Schlanke Entitäten und Domänenklassen durch Data Classes

### 2.3 Frontend Clients

Kein Teammitglied hat bis jetzt vertiefte Erfahrung im Bereich der Frontend-Entwicklung. Daher setzen wir auf das an der FHNW vermittelte Framework React, um die Clients zu implementieren. React bietet mit seinem Komponenten-Model eine einfache Abstraktionsmöglichkeit um die Anwendung sauber zu kapseln. Die Funktionalen JSX Komponenten scheinen leichtgewichtiger im Vergleich zu den HTML-Template-Ansätzen von Angular oder VueJS. Unser Ziel ist es in diesem Projekt die Kenntnisse in einem Projekt zu vertiefen und die Client-Software möglichst pur funktional zu halten.

Änderungen am Dokument werden auch im Frontend immer als Commands modelliert. Für die Verarbeitung dieser Commands und das verwalten des Zustands der Client Applikation wird redux verwendet.

### 2.4 Datenbank System

Um die kollaborativ erstellten Dokumente zu persistieren und zu verwalten setzen wir auf eine No-SQL Lösung. Das notwendige Datenmodel lässt sich elegant als *Document* abbilden. Durch den Einsatz einer No-SQL Lösung kann die Representation der Dokumente über alle Layer der Applikation gleichbleibend beibehalten werden, ohne die Notwendigkeit von ORM.

Konkret wird im Projekt MongoDB als Datenbanksystem verwendet. Wir haben uns für diese Variante aufgrund der bestehenden reaktiven Integration in das Springframework entschieden.

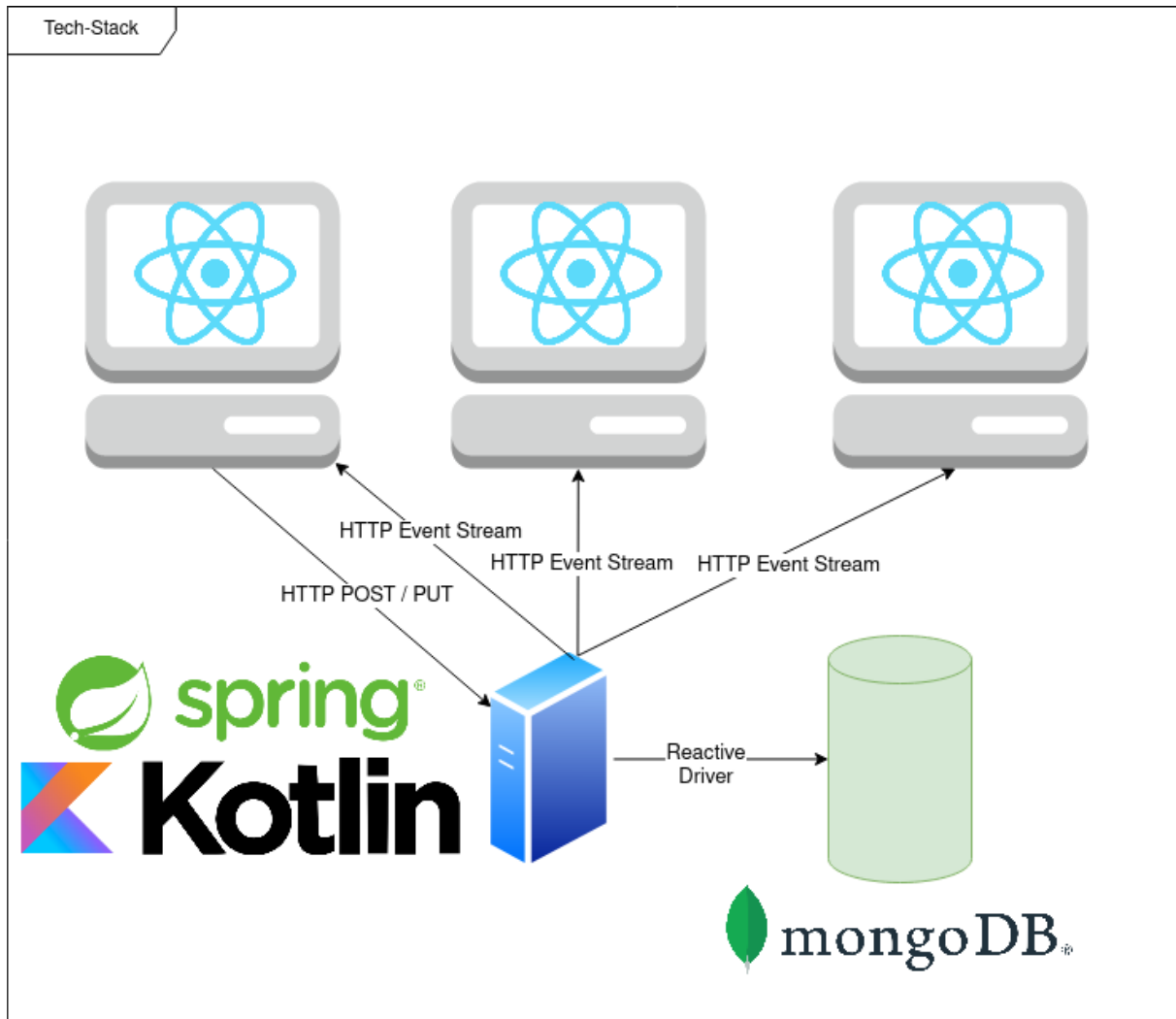


Abbildung 2.1: Technologie Stack

## **3 Systemübersicht**

### **3.1 Services**

### **3.2 Sequenz**

### **3.3 Applikationsprotokoll**

### **3.4 Benutzerverwaltung**

Es ist keine persistente Benutzerverwaltung mit Registrationsprozess implementiert. Nach erstmaligem Anmelden in der Applikation mit einem globalen Benutzer, wird ein zufälliger Author erstellt. Die Daten des Authors werden im Local Storage des Browsers gespeichert, sodass bei erneutem Öffnen der Applikation der gleiche Author wiederverwendet wird.

## 4 Frontend

Das Frontend der TeamDocument Applikation ist als React SPA entwickelt. Einmal angemeldet kann ein Benutzer an der kollaborativen Bearbeitung des Dokumentes teilnehmen.

Folgende Interaktionen sind möglich:

- Ändern des eigenen Namens
- Hinzufügen eines neuen Paragraphen
- Bearbeitung bestehender Paragraphen
- Sperren des Paragraphen an dem gerade gearbeitet wird (implizit)
- Verschieben von Paragraphen innerhalb des Dokuments
- Löschen eines bestehenden Paragraphen
- Wiederherstellen des zuletzt gelöschten Paragraphen (Hidden Feature)

Des Weiteren werden folgende Informationen auf dem UI dargestellt:

- Name des ursprünglichen Autors eines Paragraphen
- Name des Autors welcher aktiv einen Paragraphen bearbeitet.
- Highlight des eigenen aktuellen Paragraphen
- Liste mit allen Dokumentupdates in chronologischer Reihenfolge
- Avatare aller aktiven Benutzer



Abbildung 4.1: Team Document User Interface

### 4.1 Aufbau

Die UI-Elemente sind als React-Komponenten umgesetzt und hierarchisch gegliedert. Einzelne Komponenten nutzen zusätzliche Funktionalität, die in kleine Service Module ausgelagert ist.

Die Anbindung ans Backend ist mit zwei unidirektionalen Kanälen realisiert.

Der State der gesamten Applikation wird vom Redux Store bewirtschaftet.



Abbildung 4.2: Komponenten Struktur



### 4.1.1 Komponenten

Alle Komponenten sind als Functional React Components implementiert.

#### **index.js**

Das Index File ist der Eintrittspunkt für den Browser. Beim Laden der Applikation wird der Redux Store erstellt und initialisiert. Ebenfalls wird im Local Storage des Browsers geprüft, ob bereits ein User registriert ist. Ist dies nicht der Fall, so wird ein zufälliger Benutzer generiert.

#### **App.js**

Die App Komponente ist das äusserste Element, welches alle anderen Elemente hält. Wir verwenden einen BrowserRouter um zwischen dem eigentlichen Dokument und der Login-Seite zu navigieren.

#### **DocumentWrapper.js**

Wrapper um das Dokument zu schützen. Solange sich ein User noch nicht ordentlich am Backend authentifiziert hat, leitet diese Komponente den Benutzer stetig auf die Login-Page weiter.

#### **Login.js**

Login Formular, welches den Login Service verwendet. Das Formular übersetzt die eingegebenen Credentials in einen Basic Auth Header und sendet damit einen GET Request ans Backend. Bei erfolgreicher Authentifizierung wird das User Principal im Local Storage abgelegt.

#### **Error.js**

Generische Fehlermeldung, welche als modales PopUp angezeigt wird, im Falle eines fehlgeschlagenen Requests.

#### **Document.js**

Document repräsentiert den gesamten Zustand des Dokuments. In dieser Komponente werden alle Paragraphen nach Position (Ordinal) sortiert aufgelistet. Ebenfalls ist die Document Komponente der Parent für alle weiteren Elemente welche zum Dokument gehören.

#### **Paragraph.js**

Die Repräsentation eines einzelnen Paragraphen. Alle möglichen Änderungen eines einzelnen Paragraphen werden von dieser Komponente bewirtschaftet.

#### **Message.js**

Message hält eine Liste aller verarbeiteten Commands, welche vom Backend empfangen wurden.

### 4.1.2 Utils

Im Ordner Utils sind Hilfsfunktionalitäten, welche von mehreren Komponenten verwendet werden oder nicht direkt einer Komponente zugehörig sind.

### loginService.js

- login → Führt einen GET Request mit den übergebenen Credentials durch.
- logout → Entfernt das User Principal aus dem Local Storage

### nameGenerator.js

- fetchSampleName → Abfrage bei <https://akabab.github.io/starwars-api/api/id/> um einen zufälligen Autor zu erstellen

### reducer.js

Die eigentliche Fachlogik des Frontends ist im Reducer implementiert. Alle Änderungen am Zustand des Dokuments werden von einer Funktion in der Function Map durchgeführt. Die Funktionen entsprechen weitestgehend den Command Types beschrieben in Kapitel 5.5. Zusätzlich werden vom Reducer noch der Authentifizierungszustand sowie ein Fehlerzustand gesetzt.

### 4.1.3 Hooks

Hooks erlauben React Features wie State oder andere Hooks zu verwenden, ohne das eine Klasse geschrieben wird. Wir verwenden zwei Custom Hooks welche Funktionalität für mehrere Komponenten bereitstellt.

### useEventSource.js

Diese Hook erstellt beim erstmaligen Laden einer Komponente eine Referenz auf eine Polyfill Event-source. Diese ist die Schnittstelle auf welcher das Backend Updates veröffentlicht.

Die Polyfill Eventsource erlaubt im Gegensatz zum Browser-Default das HTTP-Header mit übergeben werden können. Es gelten folgende Browser Kompatibilitäten

- IE 10+, Firefox 3.5+, Chrome 3+, Safari 4+, Opera 12+
- IE 8 - IE 9: XDomainRequest is used internally, which has some limitations (2KB padding in the beginning is required, no way to send cookies, no way to use client certificates)
- It works on Mobile Safari, Opera Mobile, Chrome for Android, Firefox for Android
- It does not work on: Android Browser(requires 4 KB padding after every chunk), Opera Mini

1

### useDebounceMessages.js

Jedes einzelne Update innerhalb eines Paragraphen generiert einen POST-Request. Es hat sich gezeigt das dieser Ansatz bei mehreren Benutzern, welche gleichzeitig schnell schreiben, einen Flaschenhals bildet. In extremen Fällen haben sich einzelne Requests gegenseitig überholt. Dadurch entstand für alle anderen Teilnehmer ein Flackereffekt in den entsprechenden Paragraphen. Um die HTTP-Schnittstelle zu entlasten werden sämtliche Textupdates mit einem leichten Debounce aggregiert und so als Liste von einzelnen Commands an der Server geschickt. Das Debounce Delay ist auf 150 ms gesetzt. Dementsprechend wird die Liste mit Updates erst nach einer Eingabepause von 150 ms versendet. Je nach Tippgeschwindigkeit werden so zwischen 5 und 15 Commands aggregiert. Dies hat zu einer signifikanten Entlastung der Netzwerklast geführt.

---

<sup>1</sup><https://github.com/Yaffle/EventSource#browser-support>



Abbildung 4.3: UI-Components

## 4.2 State- und Konfliktmanagment

Der gesamte Zustand der Applikation ist im Redux Store abgebildet. Der Zustand enthält folgende Komponenten:

Key: isAuthenticated  
 Type: Boolean  
 Beschreibung: Flag für Routing zur Login-Page, falls false

Key: author  
 Type: Object id: UUID, name: String, image: String  
 Beschreibung: Repräsentation des eigenen Autors.

Key: otherAuthors  
 Type: Array  
 Beschreibung: Liste aller anderen aktiven Autoren.

Key: paragraphs  
 Type: Array  
 Beschreibung: Liste aller vorhandenen Paragraphen.

Key: messages  
 Type: Array  
 Beschreibung: Liste der vom Server erhaltenen Commands.

Key: error  
 Type: Object isPresent: Boolean, message: String  
 Beschreibung: Objekt welches einen Fehler anzeigt sofern einer vorhanden ist

Beim Laden des Frontends baut der Client die Eventsource zum Server auf. Dieser sendet anschliessend den aktuellen Zustand des Dokuments mit einem Initial Command zum Client.

Änderungen an der eigenen Kopie des Dokumentes werden direkt als Action an den Reducer dispatched. Parallel wird die Änderung als Command an den Server gesendet welcher den Zustand des Dokuments aktualisiert und an alle anderen Teilnehmer weiterleitet.

Allfällige Konflikte werden dabei vom Server detektiert und aufgelöst. Diese korrektoren werden dann von allen Clients verarbeitet.

## Ablaufdiagramm

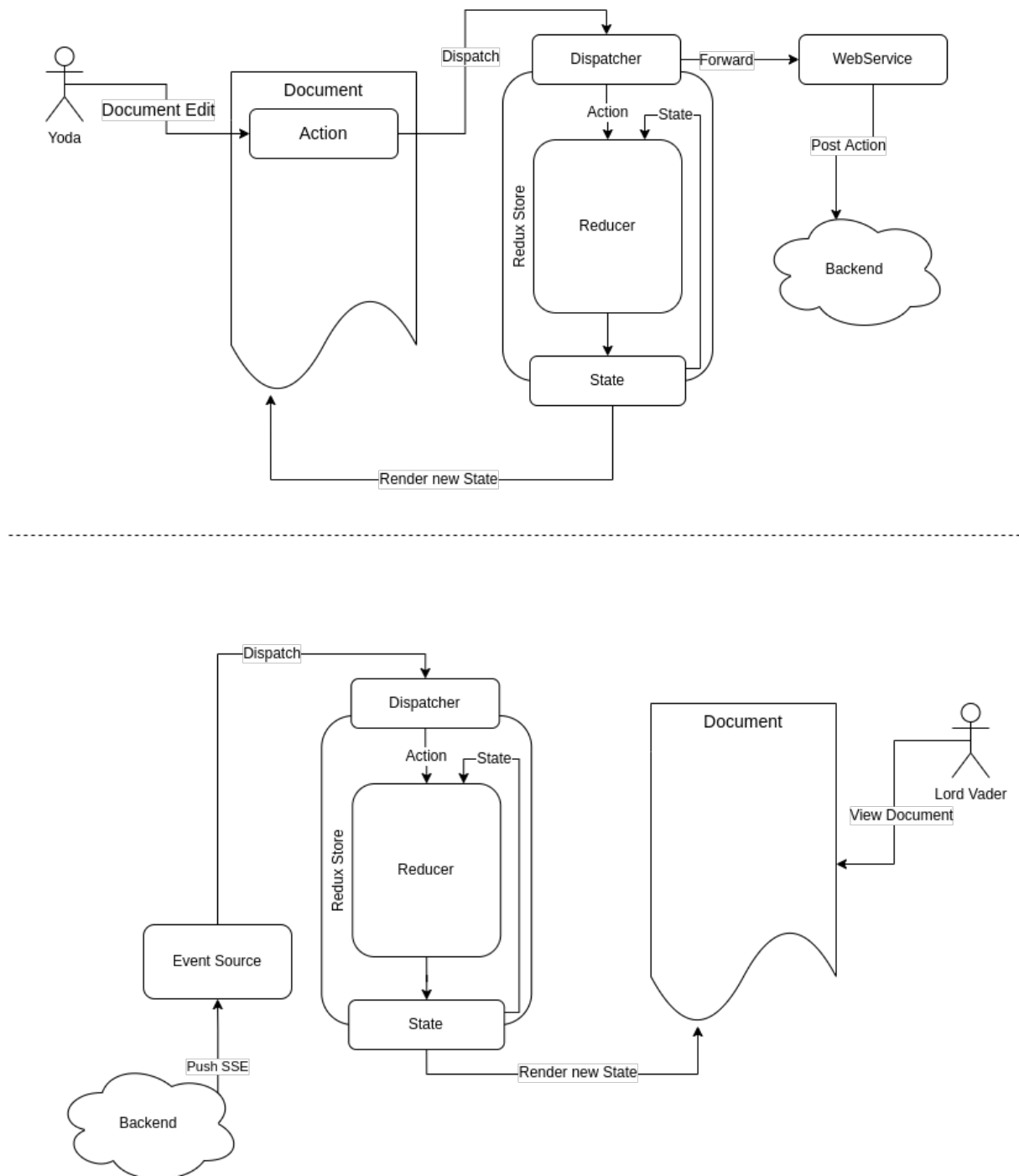


Abbildung 4.4: Datenfluss

## 4.3 Fehler Behandlung

Fachliche Fehler werden vom Client nicht behandelt. Mögliche Fehler sind noch Verbindungsprobleme zum Server oder ungültige Zustände des Servers.

Ist der Server nicht erreichbar, wird das UI blockiert, so dass keine Eingaben mehr möglich sind. Der Benutzer wird mit einer entsprechenden Fehlermeldung auf das Problem aufmerksam gemacht.

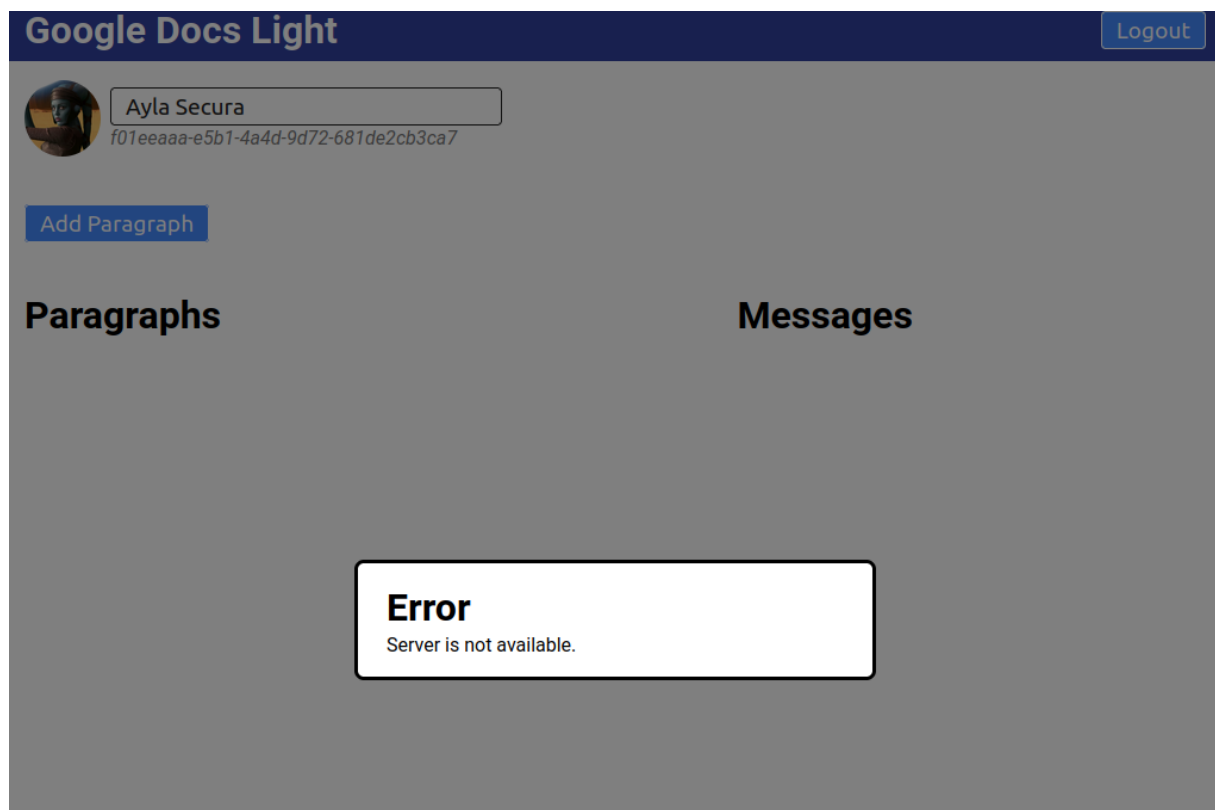
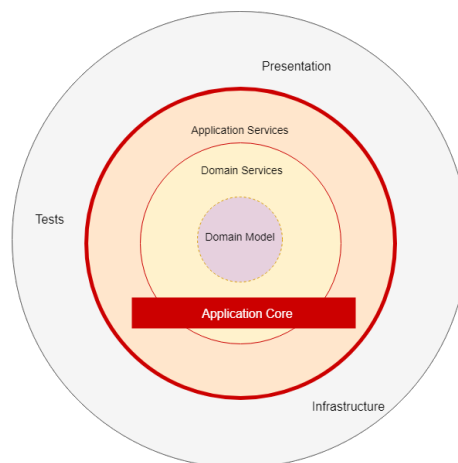


Abbildung 4.5: Fehler Meldung

## 5 Backend

### 5.1 Aufbau

Der Aufbau der Serverapplikation lehnt sich am Konzept der Onion-Architecture an. In Onion Architecture wird die Applikation in Layer aufgeteilt.



**Abbildung 5.1:** Onion Architecture

Um zu garantieren, dass keine ungewollten Abhängigkeiten zwischen Layern bestehen, können die Layer in eigene Module verpackt und Abhängigkeiten über Interfaces abstrahiert werden. Dies erhöht jedoch die interne Komplexität der Applikation. Die Umsetzung wird aufgrund der geringen Projektgrösse deshalb nicht in unabhängigen Modulen realisiert, sondern über die Package Struktur gelöst. Angesichts dieser Entscheidung wird weiter darauf verzichtet, Abhängigkeiten zwischen Modulen über Interfaces zu abstrahieren. Da es nie mehrere Implementationen einer Komponente geben wird, bringt der Einsatz von Interfaces keinen grossen Mehrwert. Bei der Implementation wird dabei konsequent darauf geachtet, die einzelnen Layer so zu halten, dass diese als eigenständige Module extrahiert werden können. Für die Verwaltung der Komponenten der Serverapplikation wird folgende Packagestruktur definiert:

```
ch.fhnw.woweb.teamdocumentserver
├── config
├── domain
├── persistence
├── service
└── web
```

**Abbildung 5.2:** Package Struktur Cloud Service

Der Domain Layer wird durch das Package domain abgebildet. Dieses beinhaltet die Domänenobjekte und darf keine Abhängigkeiten auf andere Module oder Frameworks beinhalten. Umgekehrt dürfen alle anderen Layer Abhängigkeiten auf den Domain Layer haben. Die Fachlogik der Applikation wird im Domain Service Layer implementiert. Dieser wird durch das Package service abgebildet. Das Package Service beinhaltet alle Komponenten, welche die Domänenobjekte verwalten oder den internen Zustand der Applikation führen. Der Layer Application Services bildet die Brücke zwischen externer Infrastruktur und Domain Services. Er ist in den Packages persistence und web beinhalten abgebildet. Dabei definiert das Package persistence Services, welche für Interaktion mit der Datenbank verwendet werden. Das Package web definiert die HTTP-Endpunkte, welche für die Kommunikation mit dem Frontend des Systems verwendet werden. Letztlich beinhaltet das Package config die technische Konfiguration der Applikation.

## 5.2 API

Die Backendapplikation bietet eine HTTP-Schnittstelle, welche von Frontendapplikationen verwendet werden kann. Die Schnittstelle ermöglicht es, sich im System anzumelden, Dokumente zu laden und Änderungen an Dokumenten zu laden und speichern. Um diese Funktionalität zu ermöglichen, bietet die Schnittstelle die drei Bereiche "Authentication", "Document" und "Message".

### 5.2.1 API Authentication

Beschreibung: Authentifizierung mit Basic Auth  
Endpunkt: /api/v1/authentication  
Methode: GET  
Headers: Authentication: Basic  
Response Code: 200, 401 oder 500  
Response Body: application/json

### 5.2.2 API Document

Beschreibung: Dokument laden und Updates abonnieren  
Endpunkt: api/v1/document  
Methode: GET  
Headers: Authentication: Basic  
X-ClientId: text  
Response Code: 200, 401 oder 500  
Response Body: text/event-stream

### 5.2.3 API Message

Beschreibung: Änderung an Dokument vornehmen  
Endpunkt: /api/v1/message  
Methode: POST  
Headers: Authentication: Basic  
Content-Type: application/json  
Body: DocumentCommand  
Response Code: 200, 401 oder 500

Beschreibung: Zuletzt gelöschten Paragraphen wiederherstellen  
Endpunkt: /api/v1/message/restore  
Methode: POST  
Headers: Authentication: Basic  
Response Code: 200, 401 oder 500

Beschreibung: Dokument zurücksetzen  
Endpunkt: /api/v1/message/restore  
Methode: DELETE  
Headers: Authentication: Basic  
Response Code: 200, 401 oder 500



## 5.3 Komponenten

### 5.3.1 Package Domain

Im Zentrum der Domäne stehen die beiden Klassen DocumentCommand und Document.

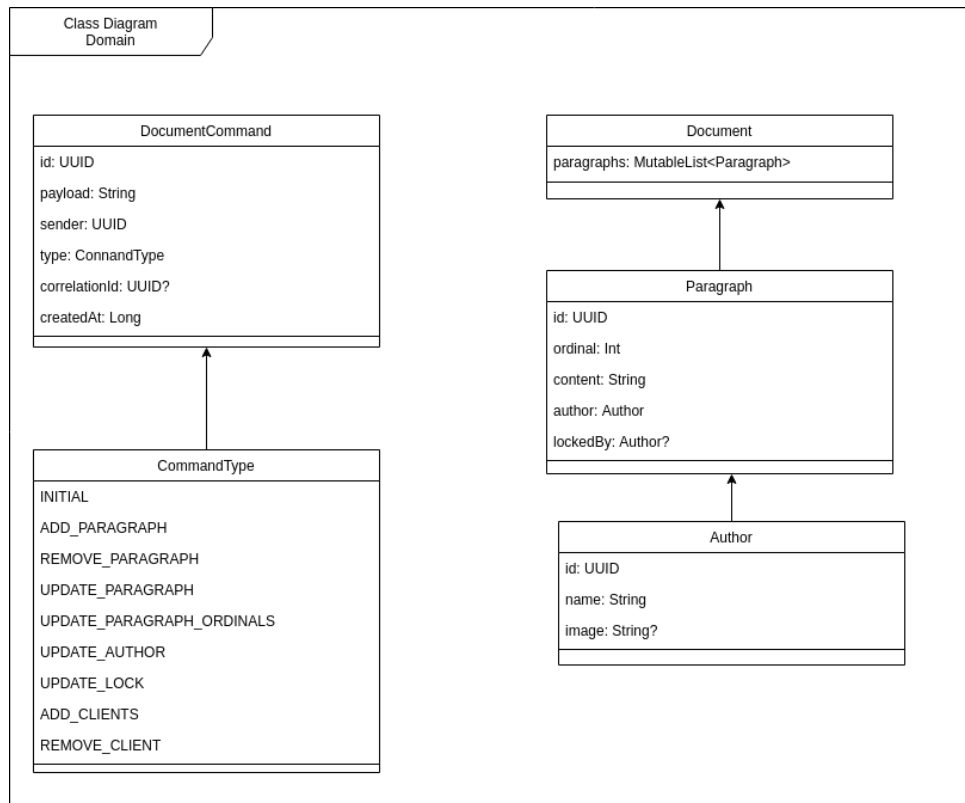


Abbildung 5.3: Klassendiagramm Domain

#### Document

Eine Instanz der Klasse Document repräsentiert den aktuellen Zustand eines Dokuments. Dieser Zustand wird in der Serverapplikation geführt und verwaltet. Ein Document besteht im Wesentlichen aus einer Liste von Paragraphs. Diese Liste kann mutiert, aber nicht ersetzt werden.

#### Paragraph

Eine Instanz der Klasse Paragraph repräsentiert einen Abschnitt in einem Dokument. Jeder Paragraph wird durch eine UUID identifiziert. Ein Paragraph definiert weiter ein Attribut Content welches den Textinhalt des Abschnitts beinhaltet und ein Attribut Ordinal, welches die Position des Abschnitts im Dokument darstellt. Weiter ist jeder Paragraph einem Author zugewiesen. Letztlich hat ein Paragraph ein Optionales Attribut lockedBy. Dieses kann entweder leer (NULL) sein oder einen Author beinhalten. Dieses Feld kann von Consumern der API verwendet werden um das bearbeiten eines Paragraphen zu erlauben oder verbieten.

#### Author

Eine Instanz der Klasse Author repräsentiert einen Benutzer, der an einem Dokument mitarbeitet. Jeder Author wird durch eine UUID identifiziert und muss einen Namen definieren. Weiter besitzt ein Author ein optionales Attribut image. Darin kann die URL zu einem benutzerbild abgespeichert werden

### **DocumentCommand**

Eine Instanz der Klasse DocumentCommand stellt eine Änderung am Zustand einer Document-Instanz dar. DocumentCommands werden als einzige Entität, Persistenzschicht abgelegt werden. Damit DocumentCommands eindeutig identifiziert werden können, beinhaltet jede Instanz ein Attribut vom Type UUID. Diese id wird auch als Identifikator in der MongoDB verwendet.

Die Änderungen welche ein DocumentCommand darstellt, werden über die Attribute payload und type definiert. Die payload hat den Typ String und beinhaltet JSON-Serialisierten des neuen Zustands, der Änderung die vorgenommen werden soll. Das Feld type beinhaltet einen Wert aus der Enum CommandType. Dieser Wert kann in den Domänenservices verwendet werden um die Payload korrekt zu Deserialisieren und die nötigen Änderungen am Document vorzunehmen.

Das Optionale Feld correlationId kann entweder NULL oder eine UUID beinhalten. Eine allfällige UUID zeigt immer auf die Id eines anderen DocumentCommands, welcher mit dem aktuellen Command zusammenhängt. Dadurch wird es möglich die Identifikation der Payload eines Commands zu verwenden, ohne die Payload jedesmal deserialisieren zu müssen.

### **CommandType**

Bei CommandType handelt es sich um eine Enum. Diese Enum beinhaltet alle Arten von DocumentCommands, welche im System bekannt sind. CommandTypes werden als ihr String Wert auf DocumentCommands persistiert.

## **5.3.2 Package Web**

### **AuthenticationController**

Die Klasse AuthenticationController implementiert einen RestController. Dieser stellt einen einzelnen GET Endpunkt zur Verfügung, über welchen sich Benutzer mittels Basic Authentication anmelden können.

### **CommandController**

Die Klasse CommandController implementiert einen RestController. Dieser Controller bietet zwei POST Endpunkte zur Verfügung. Über den ersten Endpoint kann eine Liste von DocumentCommands an den Server gesendet werden. Der Endpoint übergibt diese Liste von Commands an den DocumentService. Diese wenden die Änderungen am Zustand des Dokuments an und leiten die Änderungen an andere Teilnehmer weiter. Über den zweiten Endpoint kann ein gelöschter Paragraph wiederhergestellt werden. Die entsprechende Fachlogik wird an den DocumentService delegiert.

### **DocumentStreamUpdateController**

Die Klasse DocumentStreamUpdateController implementiert einen RestController. Dieser Controller ermöglicht es, den aktuellen Zustand eines Dokumentes zu laden und Änderungen am Dokument zu abonnieren. Der Endpoint, welcher dazu zur Verfügung steht erwartet, dass der Custom Header "X-ClientId" gesetzt ist. Dieser muss einen Stringwert beinhalten, welchen den Author der die Daten anfragt identifiziert. Das Laden des Dokuments und das Erstellen der Abonniierung wird an den DocumentService delegiert. Als Rückgabotyp wird "Flux<DocumentCommand>" verwendet. Dadurch ist es möglich den Status des Documents und alle folgenden Änderungen in einem Stream zurückzugeben.

### 5.3.3 Package Services

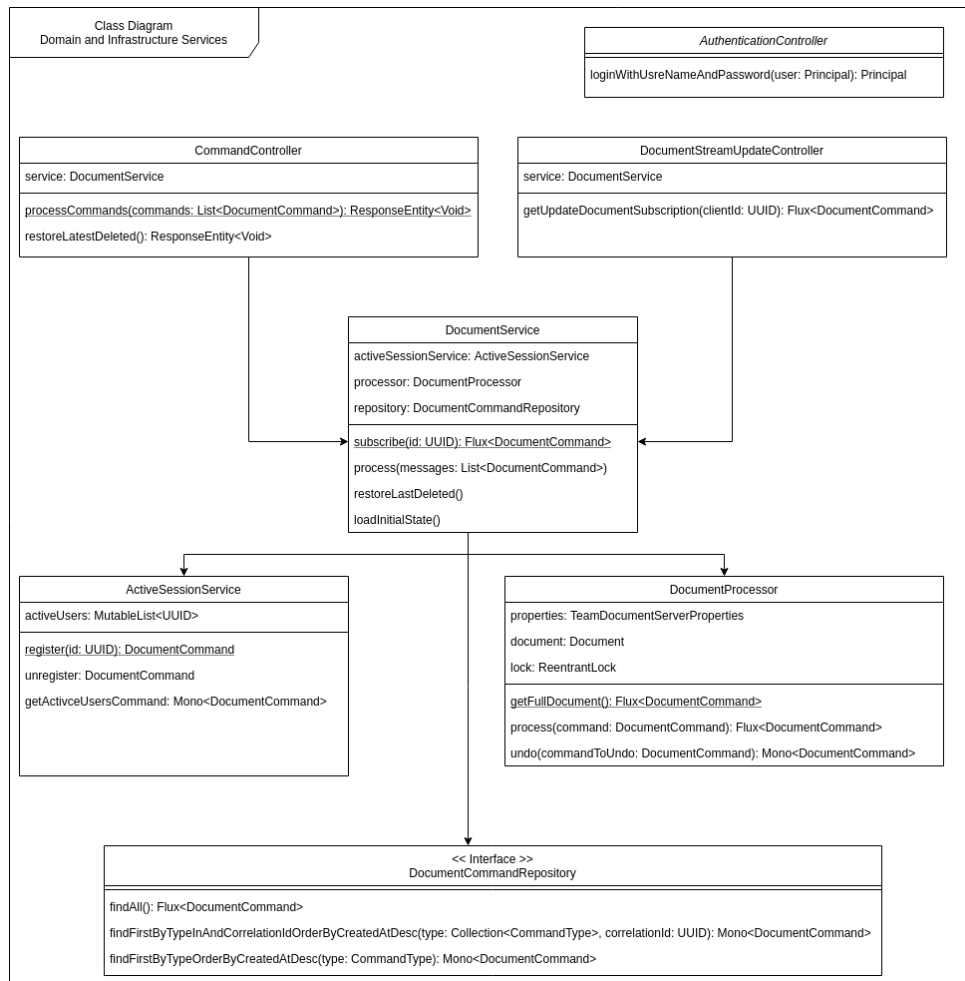


Abbildung 5.4: Klassendiagramm Services

#### DocumentService

Die Klasse **DocumentService** ist dafür Verantwortlich erhaltene Anfragen für Dokumente und Änderungen an Dokumenten zu verarbeiten. Sie delegiert die entsprechende Fachlogik Klassen **ActiveSessionService**, **DocumentProcessor** und **DocumentCommandrepository**.

Die Methode `process` erlaubt es, eine Liste von **DocumentCommands** zu verarbeiten. Die Methode `subscribe` erlaubt es, einen Stream des aktuellen Zustands des Dokuments und aller künftigen Änderungen an einem Dokument anzufragen. Beide Abläufe werden in Kapitel 5.4 beschrieben.

#### DocumentProcessor

Die Klasse **DocumentProcessor** führt den Zustand des Dokuments, welches mit der Applikation verwaltet wird. Er ist dafür Verantwortlich, Änderungen an diesem Dokument vorzunehmen. Dazu besitzt Sie ein Attribut `document` vom gleichnamigen Typ. Über die Methode `process` kann ein einzelner **DocumentCommand** angewendet werden. Der Processor verarbeitet denm Command anhand des gesetzten `CommandTypes`. Dabei muss er allfällige Konflikte erkennen und auflösen. Nach der Verarbeitung des Commands werden alle Änderungen und Konfliktlösungen als **DocumentCommands** zurückgegeben.

### **ActiveSessionService**

Die Klasse `ActiveSessionService` führt den Zustand der aktiven Nutzer einer Session. Dazu führt die Klasse eine Liste der Identifikatoren aller aktiven Nutzer einer Session. Der Service bietet Methoden um die aktiven Benutzer auszulesen, einen neuen Benutzer zu registrieren und einen Benutzer zu entfernen.

### **5.3.4 Package Persistence**

#### **DocumentCommandRepository**

Das Interface `DocumentCommandRepository` erweitert das Interface `ReactiveCrudRepository` von Spring. Es kann damit verwendet werden um Create, Read, Update und Delete Optionen für `DocumentCommands` in der angebundenen MongoDB auszuführen.

### **5.3.5 Konfiguration**

#### **Spring**

Alle Klassen im Package `SServiceBind` mit der Spring-Boot-Annotation `"@Service"` versehen. Sie können damit automatisch von Spring-Boot instanziiert werden und stehen Sie in Spring Beans zur Verfügung und können über Constructor-Injection verwendet werden.

Alle Klassen im Package `"webBind"` mit der Spring-Boot-Annotation `"@RestController"` versehen. Sie können damit automatisch von Spring-Boot instanziiert werden.

#### **application.yml**

Die Datei `application.yml` beinhaltet die konfigurierbaren Werte der Serverapplikation. Dies beinhaltet die Konfiguration der angebundenen MongoDB, Referenzen zu Umgebungsvariablen mit User Credentials und Logging Konfiguration.

#### **TeamDocumentServerProperties**

Die Konfigurationsklasse `TeamDocumentServerProperties` ist mit der Annotation `"@ConfigurationProperties(prefix = 'teamdocument')"` versehen. Sie kann wo nötig in den Serviceklassen verwendet werden, um auf Werte aus dem `application.yml` zuzugreifen.

#### **WebConfig**

Die Klasse `WebConfig` beinhaltet die Konfiguration für `SpringSecurity`.

## 5.4 Abläufe

### 5.4.1 Dokument laden und Änderungen abonnieren

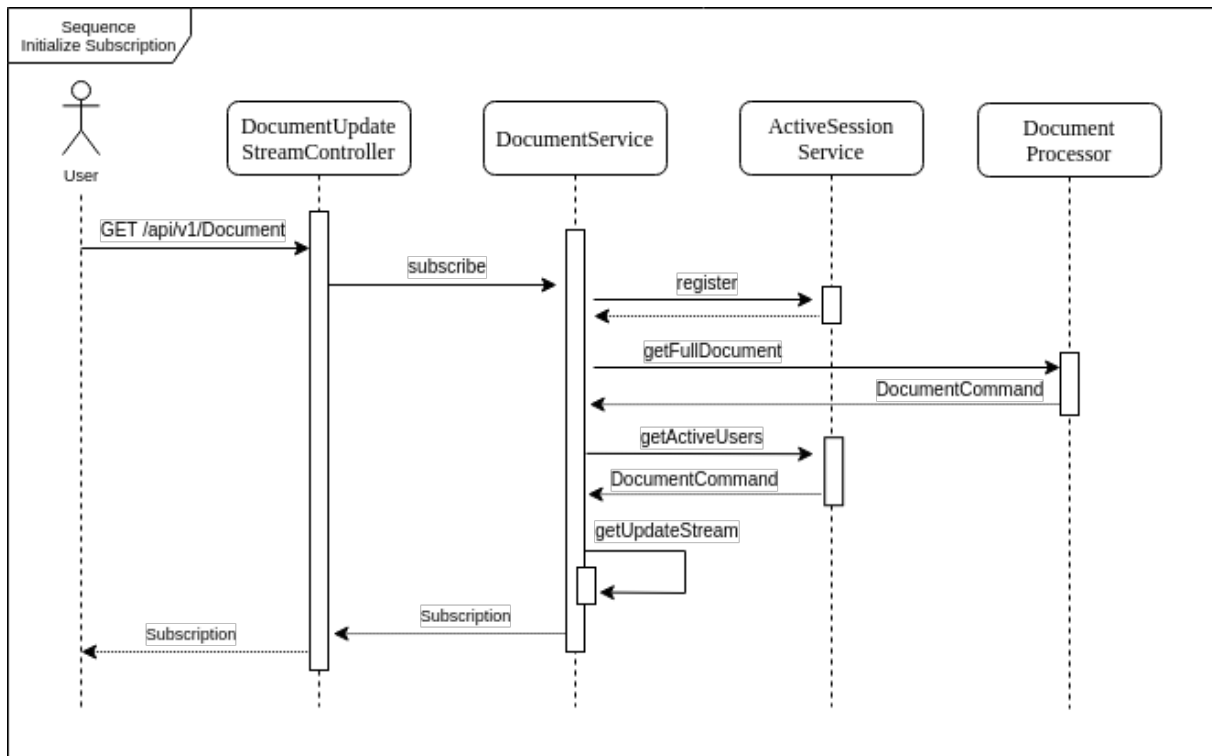


Abbildung 5.5: Sequenzdiagramm Subscription erstellen

Die Komponente `DocumentUpdateStreamController` erlaubt es ein Dokument zu laden und Änderungen an diesem Dokument zu abonnieren. Dazu bietet der `RestController` einen Endpunkt, welcher den ein Resultat vom Typ `Flux<DocumentCommand>` zurückgibt. Nachdem eine Anfrage beim Controller eingegangen ist, delegiert dieser die Erstellung des Flux an den `DocumentService`. Dieser registriert den Client hinter der Subscription beim `ActiveSessionService` und trägt anschliessend Informationen aus drei Quellen zusammen. Zuerst wird der aktuelle Stand des Dokuments beim `DocumentProcessor` angefragt. Anschliessend wird eine Liste aller aktiven Clients des Dokuments aus dem `ActiveSessionService` geladen. Letztlich wird eine Subscription für Änderungen am Dokument erstellt. Der `DocumentService` führt dazu eine Instanzvariable vom Typ Sink. Alle Änderungen werden nach der Anwendung in diesen Sink geschrieben. Eine Subscription auf diesem Sink beinhaltet damit alle Änderungen, welche vorgenommen wurden. Die Informationen zum initialen Stand des Dokuments, aktiven Clients und die Subscription auf Änderungen werden anschliessend in einem einzelnen Flux zusammengefasst und zurückgegeben. Nachdem der Flux geschlossen wird, wird die Registrierung des aktiven Benutzers im `ActiveSessionService` wieder entfernt.

### 5.4.2 Änderungen verarbeiten

Die Komponente `CommandController` erlaubt es Änderungen an einem Dokument vorzunehmen und zu persistieren. Der Controller bietet dazu einen HTTP-Endpunkt, über welchen ein JSON-Serialisierter `DocumentCommand` übergeben werden kann. Ein `DocumentCommand` definiert unter anderem einen type, welcher ein Wert der Enum `CommandType` sein muss und eine Payload vom Typ `String`. Die Payload ist wiederum ein JSON-Serialisiertes Objekt. Die Verarbeitung eines `DocumentCommands` wird vom Controller an den `DocumentService` delegiert. Dieser übergibt den Command wiederum an den `DocumentProcessor`. Dieser wendet Änderungen an und löst Konflikte auf. Die übergebenen Commands und

allfällige Commands zur Konfliktlösung werden vom DocumentProcessor zurückgegeben. Der DocumentService übergibt diese an das DocumentCommandRepository zur persistierung in der Datenbank. Anschliessend werden die persistierten Commands über die Instanzvariable vom Typ Sink veröffentlicht. Registrierte Clients haben eine Subscription auf diesen Sink und werden so über die Änderungen informiert.

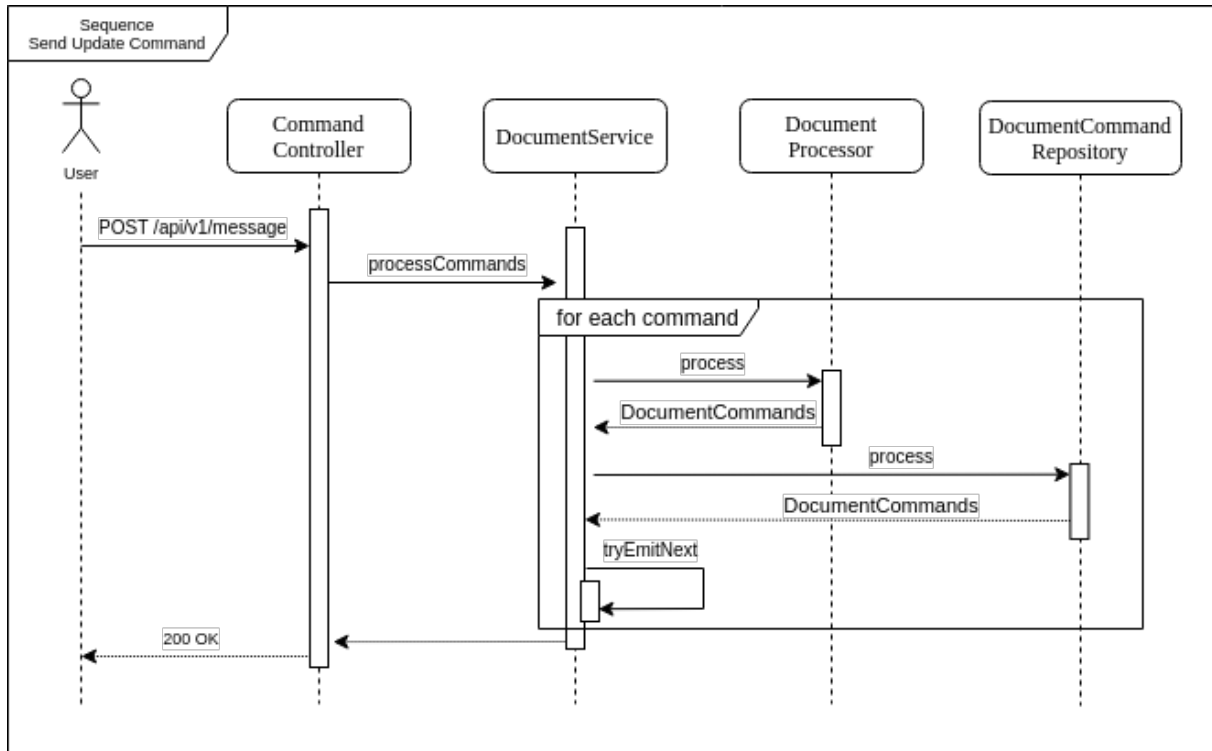


Abbildung 5.6: Sequenzdiagramm Command verarbeiten

### 5.4.3 Fehlerbehandlung

Fachliche Fehler, die während der Verarbeitung von Änderungen an einem Dokument auftreten, werden innerhalb des DocumentProcessors behandelt. Dabei werden immer zusätzliche Änderungen generiert, welche Konflikte lösen. Die Verarbeitung wird nie durch das Werfen von Exceptions unterbrochen. Es ist möglich, dass die Verarbeitung einer Änderung oder das Veröffentlichen von Änderungen wegen einem technischen Fehler fehlschlägt.

Wenn eine Exception beim Verarbeiten einer Änderung auftritt, wird die Verarbeitung abgebrochen und die Exception weiter geworfen. Exceptions werden in den Controller Klassen abgefangen. Im Fehlerfall wird dort eine ResponseEntity mit Statuscode 500 zurückgegeben.

Wenn eine Exception beim Veröffentlichen einer Änderung auftritt, wird die Subscription geschlossen. Dadurch wird die Verbindung des betroffenen Clients getrennt. Es liegt in der Verantwortung des Clients die Verbindung erneut zu öffnen und den aktuellen Stand des Dokuments zu laden.

## 5.5 Zustands- und Konfliktmanagement

Um sicherzustellen, dass ein Dokument auf allen Clients einen konsistenten Zustand hat, wird der Zustand des Dokuments in der Serverapplikation geführt. Änderungen von Clients müssen an den Server gesendet werden. Dieser wendet die Änderungen auf dem Dokument an und leitet die Änderungen an alle anderen Clients weiter. Während der Anwendung von Änderungen, ist es in der Verantwortung der Serverapplikation, sicherzustellen, dass das Dokument in einem Konsistenten Zustand bleibt. Wenn nötig, erstellt es dazu zusätzliche Änderungen und veröffentlicht diese ebenfalls an alle Clients.

Die entsprechende Fachlogik ist in den Klassen `DocumentService` und `DocumentProcessor` implementiert. Dabei ist die Klasse `DocumentProcessor` für die Zustandsverwaltung des Dokuments verantwortlich.

### 5.5.1 Grundsatz

Der `DocumentProcessor` hat zwei private Instanzvariablen, die für das Zustandsmanagement relevant sind. Die Variable **document** ist vom gleichnamigen Typ. Diese Document-Instanz stellt die "Source Of Truth" für den Zustand des Dokuments dar. Ein Document besteht im Wesentlichen aus einer Liste von Paragraphen. Diese Liste wird mit `synchronizedList(mutableListOfParagraphs())` initialisiert. Dadurch wird sichergestellt, dass der Zugriff für die Liste zwischen der Verarbeitung einzelner Commands synchronisiert ist. Die Variable **lock** hat den Typ `ReentrantLock`. Die Verarbeitung einiger DocumentCommands darf nicht parallel passieren, weil dadurch Konflikte ausgelöst werden können. Die Verarbeitung solcher DocumentCommands wird durch die Verwendung dieses `ReentrantLocks` gesperrt.

### 5.5.2 Zustandsänderungen

Alle Änderungen werden als Instanzen der Klasse `DocumentCommand` an den Server übermittelt. Nachfolgend wird beschrieben, welche Arten von DocumentCommands unterstützt sind, wie diese verarbeitet werden und wie mit möglichen Konflikten umgegangen wird.

Ein `DocumentCommand` mit Typ **INITIAL** stellt den vollständigen Zustand eines Dokuments dar. Er beinhaltet als Payload eine Liste der Paragraphen in diesem Dokument. INITIAL Commands werden im Backend generiert, um den vollständigen Zustand eines Dokuments zu veröffentlichen. Eingehende INITIAL Commands werden im `DocumentProcessor` nicht angewendet.

Ein `DocumentCommand` mit Typ **ADD\_PARAGRAPH** fügt einen neuen Abschnitt zum Dokument hinzu. Er beinhaltet als Payload einen einzelnen Paragraphen. Dieser Paragraph wird im `DocumentProcessor` deserialisiert und der Liste von Paragraphen im Dokument hinzugefügt. Es ist möglich, gleichzeitig zwei Commands eingehen, welche an derselben Stelle im Dokument einen Paragraphen einfügen möchten. Deshalb wird hier sichergestellt, dass die Ordinalnummern aller Paragraphen korrekt sind und keine Nummer doppelt vorkommt. Während diese Korrektur vorgenommen wird, dürfen keine anderen Änderungen möglich sein. Deshalb wird die Verarbeitung eines **ADD\_PARAGRAPH** Commands mit dem `ReentrantLock` versehen. Nach der Verarbeitung werden der **ADD\_PARAGRAPH** und **UPDATE\_PARAGRAPH\_ORDINALS** Command zurückgegeben, damit Sie veröffentlicht werden können.

Ein `DocumentCommand` mit Typ **REMOVE\_PARAGRAPH** entfernt einen Paragraphen aus dem Dokument. Er beinhaltet als Payload die `UUID`, des zu entfernenden Abschnitts. Bei der Verarbeitung wird der Abschnitt mit der gegebenen Id entfernt. Ist der Abschnitt bereits entfernt, wird kein Fehler geworfen. Nachdem ein Paragraph entfernt wurde, muss sichergestellt werden, dass es keine Lücke in den Ordinal Nummern der Paragraphen gibt. Deshalb wird auch hier sichergestellt, dass die Ordinalnummern aller Paragraphen korrekt sind. Anschliessend werden die **REMOVE\_PARAGRAPH** und **UPDATE\_PARAGRAPH\_ORDINALS** zurückgegeben. Um sicherzustellen, dass die Ordinalnummern korrekt gesetzt werden, ist auch diese Verarbeitung mit dem `ReentrantLock` abgeschlossen.

Ein DocumentCommand mit Typ **UPDATE\_PARAGRAPH** aktualisiert den Textinhalt eines Abschnitts. Er beinhaltet als Payload einen einzelnen Paragraphen. Bei der Verarbeitung wird der relevante Abschnitt im Dokument gefunden und dessen Inhalt überschrieben. Hier wird bewusst kein explizites Konfliktmanagement betrieben. Paragraphen dürfen nur bearbeitet werden, wenn der Benutzer den Paragraph für sich gesperrt hat. Da ein Paragraph immer nur von einem Benutzer gesperrt sein kann und Updates in derselben Reihenfolge wie sie geschehen eingehen, können diese Updates immer angewendet werden. Es ist möglich, dass sich das Sperren eines Paragraphen von zwei Benutzern überschneidet. Am Ende darf aber immer nur ein Benutzer den Paragraphen gesperrt haben. In diesem Fall ist es das gewünschte Verhalten, dass die Änderungen dieses Benutzers alle anderen Änderungen am selben Abschnitt überschreiben.

Ein DocumentCommand mit Typ **UPDATE\_PARAGRAPH\_ORDINALS** aktualisiert die Ordinalnummern von Abschnitten. Er beinhaltet als Payload eine Liste von Paragraphen. Bei der Verarbeitung dieses Commands werden die Ordinalnummern aller Abschnitte mit den Ordinalnummern aus der Payload überschrieben. Anschliessend wird sichergestellt, dass die Ordinalnummern aller Paragraphen korrekt sind und keine Nummer doppelt vorkommt. Es werden darauf der erhaltene Command und allfällige Commands zur Konfliktlösung zurückgegeben. Um sicherzustellen, dass die Ordinalnummern korrekt gesetzt werden, ist auch diese Verarbeitung mit dem ReentrantLock abgeschlossen.

Ein DocumentCommand mit Typ **UPDATE\_AUTHOR** aktualisiert den Namen eines Authors, der das Dokument bearbeitet. Er beinhaltet als Payload eine Author-Instanz. Der Name dieses Authors wird auf allen Abschnitten im Dokument aktualisiert. Anschliessend wird der Command zurückgegeben. Es wird hier kein explizites Konfliktmanagement betrieben, da ein Benutzer immer nur an genau einem Gerät arbeiten kann. Sollte derselbe Benutzer auf mehreren Clients verwendet werden und gleichzeitig den Namen ändern, können sich diese Änderungen überschreiben. In diesem Fall wird das zuletzt gesendete Update angewendet und veröffentlicht. Damit ist der Zustand auch bei Konflikten konsistent.

Ein DocumentCommand mit Typ **UPDATE\_LOCK** erlaubt es einen Abschnitt durch einen Benutzer zu ent-/sperren. Dies wird in den Clients verwendet, um sicherzustellen, dass nur ein Benutzer gleichzeitig an einem Paragraph arbeiten kann. Die Payload dieses Commands beinhaltet als Payload den Paragraphen, der gesperrt werden soll. Dieser Paragraph wird in der Liste von Paragraphen gefunden und durch setzen des `lockedBy` Attributs gesperrt. Ist auf der Payload kein `lockedBy` Attribut gesetzt, wird die Sperre entfernt. Dabei wird sichergestellt, dass eine Sperre nur durch den Benutzer der sie erstellt hat entfernt werden kann. Versucht ein anderer Benutzer, die Sperre aufzuheben, wird die Verarbeitung abgebrochen und ein Command welcher die Sperre zurücksetzt zurückgegeben. Damit sich das Sperren von Paragraphen zwischen Benutzern nicht überschneiden kann, ist die Verarbeitung dieses Commands mit dem ReentrantLock versehen.

Ein DocumentCommand mit Typ **ADD\_CLIENTS** teilt mit, dass ein neuer Bearbeiter am Dokument existiert. Dieser Command führt zu keiner Änderung am Dokument und wird im DocumentProcessor nicht verarbeitet. Er wird aber zurückgegeben, damit er an alle anderen Clients weitergegeben werden kann.

Ein DocumentCommand mit Typ **REMOVE\_CLIENT** teilt mit, dass die Verbindung eines Bearbeiters getrennt wurde. Bei der Verarbeitung dieses Commands werden alle Paragraphen, welche durch diesen Bearbeiter gesperrt waren entsperrt. Dadurch wird sichergestellt, dass Paragraphen nicht gesperrt sind wenn ein Benutzer den Client beendet oder die Verbindung abbricht.



## **6 Testing**

### **6.1 Frontend**

### **6.2 Backend**

#### **6.2.1 Unit Tests**

Sämtliche Service- und Controller Klassen werden mit Unit-Tests getestet. Dazu wird das Framework JUnit verwendet. Die Unit Tests testen jeweils genau eine Klasse. Sämtliche Abhängigkeiten auf andere Services werden mit dem Framework Mockito gemocked.

#### **6.2.2 Lasttests**

Neben den einfachen Unit Tests wurden für den DocumentProcessor Lasttests implementiert. Diese stellen sicher, dass der DocumentProcessor Änderungen auch unter grössere Last schnell verarbeitet und dabei einen konsistenten Zustand im Dokument erstellt. In diesen Lasttests werden drei Benutzer simuliert welche parallel je 512 DocumentCommands verarbeiten lassen. Dabei wird vor jedem Verarbeitungsschritt eine zufällige Verzögerung von max. einer Sekunde eingebaut. Der Test stellt sicher, dass im Schnitt nicht mehr als vier Millisekunden für die Verarbeitung eines Commands verwendet. Der Test prüft weiter, dass das Dokument nach der Verarbeitung den erwarteten Zustand hat.

#### **6.2.3 Application Tests**

Mit der Testklasse TeamDocumentServerApplicationTests wird die Backendapplikation als Ganzes getestet. Dazu wird der gesamte Application Context hochgefahren. Anschliessend werden DocumentCommands direkt über die Controller Klassen verarbeitet. Dabei wird geprüft, dass Änderungen korrekt angewendet und veröffentlicht werden.

In den Application Tests werden zudem einfache Lasttests ausgeführt. Gleich wie bei den Lasttests auf dem DocumentProcessor werden hier drei Benutzer simuliert welche parallel je 512 DocumentCommands verarbeiten lassen. Anschliessend wird sichergestellt, dass im Schnitt nicht mehr als 40 Millisekunden für die Verarbeitung eines Commands verwendet wird. Es wird weiter geprüft, dass das Dokument nach der Verarbeitung in einem Konsistenten zustand ist.

### **6.3 End to End Test**

## 7 Deployment

Die Applikation ist auf einen virtuellen Ubuntu Server bei Switch Tube deployt. Für Frontend und Backend sind eigene Docker Images erstellt. Für die Datenbank verwenden wir das Offizielle MongoDB Image. Die Container werden mit Docker-Compose verwaltet.

### Frontend

Das Frontend-Image wird mit einem Multi-Stage Dockerfile gebaut. Der Builder ist ein Node Container welcher den Production build erstellt. Diese Artefakte werden in einen NGINX Container kopiert.

NGINX dient dabei als Webserver für das Frontend und gleichzeitig als Reverse-Proxy. Alle API Anfragen werden so durch den NGINX Server entgegengenommen und zum internen Dockernetzwerk weitergeleitet.

### Backend

Der Spring Server wird auf Basis des OpenJdk 11 Images gebaut. Spring Webflux verwendet per Default den Reactor Netty Webserver. Die Anbindung zur MongoDB ist direkt in den Application Properties konfiguriert.

### Protokolle

Gegen die Öffentlichkeit präsentiert der NGINX Webserver TLS Zertifikate von Let's Encrypt. Alle eingehenden HTTP Anfragen auf Port 80 werden weitergeleitet nach Port 443.

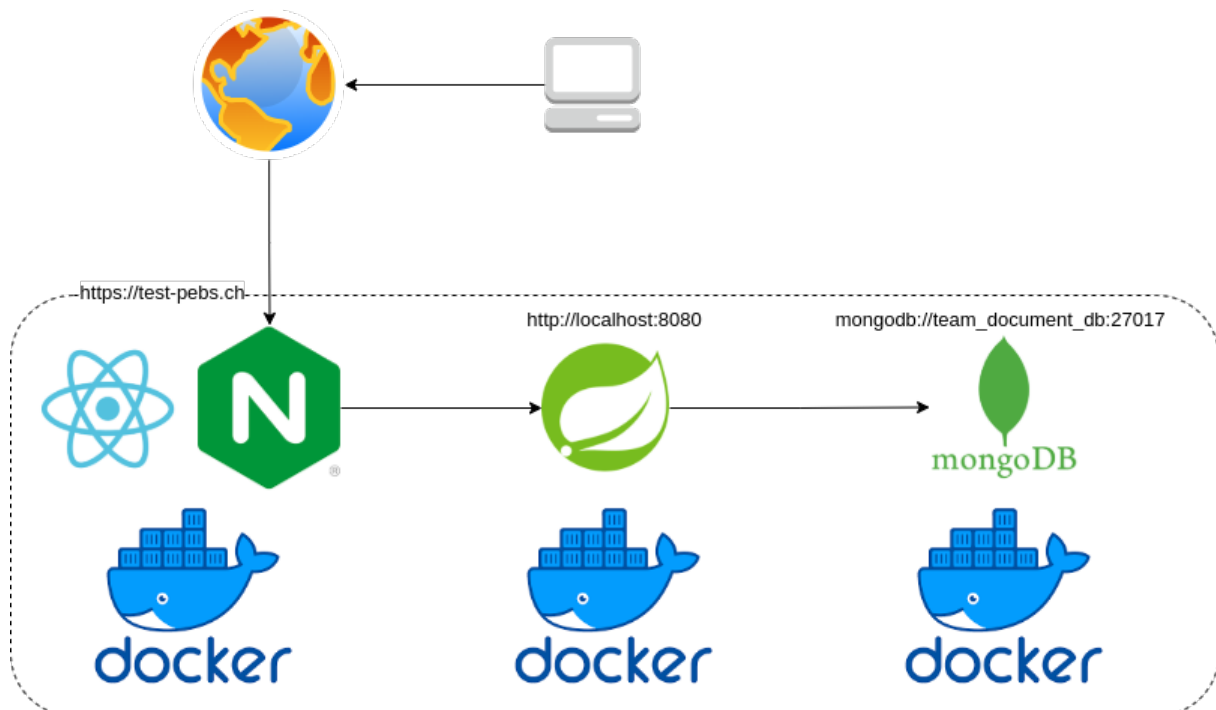


Abbildung 7.1: Deployment Übersicht

## **8 Ausblick**

## **9 Fazit**

## Abbildungsverzeichnis

2.1	Technologie Stack . . . . .	3
4.1	Team Document User Interface . . . . .	5
4.2	Komponenten Struktur . . . . .	6
4.3	UI-Components . . . . .	9
4.4	Datenfluss . . . . .	11
4.5	Fehler Meldung . . . . .	12
5.1	Onion Architecture . . . . .	13
5.2	Package Struktur Cloud Service . . . . .	13
5.3	Klassendiagramm Domain . . . . .	15
5.4	Klassendiagramm Services . . . . .	17
5.5	Sequenzdiagramm Subscription erstellen . . . . .	19
5.6	Sequenzdiagramm Command verarbeiten . . . . .	20
7.1	Deployment Übersicht . . . . .	24