Fachhochschule Nordwestschweiz
Hochschule für Technik

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

1 Ausgangslage

”Im Modul Workshop Distributed Software Systems (DSS) und Workshop Web (WOWEB) werden die Themen der Vertiefungsrichtung Verteilte Software Systeme und der Vertiefungsrichtung Web in einem durchgehenden Beispiel angewendet und umgesetzt.”[1] Im Rahmen dieses Moduls soll eine ”light” Version von Google Docs umgesetzt werden. Diese Lösung soll kollaboratives Arbeiten an einem einzelnen Dokument erlauben.

Eine zentrale Anforderung an das System ist die konsistente und verzögerungsfreie Darstellung des Dokuments auf mehreren Clients. Um diese Herausforderung zu lösen, müssen Applikationsprotokolle und Konzepte definiert werden, welche es erlaubt den Zustand eines Dokumentes zu verwalten und allfällige Konflikte zu lösen. Es müssen Technologien und Kommunikationsprotokolle gewählt werden, welche es ermöglichen diese Konzepte umzusetzen.

In den folgenden Kapiteln werden die erarbeiteten Konzepte und deren Implementierung vorgestellt. Dabei werden die gewählten Technologien und die umgesetzten Komponenten beschrieben.

[1] Zitat Aufgabenstellung: https://github.com/FHNW-WODSS-FS22/team-document-KZellweger/blob/master/readme.md

2 Systemübersicht

2.1 Lösungsstrategie

Um die verteilte Dokumentenbearbeitung zu ermöglichen, verwenden wir eine Variante von Eventsourcing. Dabei sollen die alle Änderungen am Dokument als einzelne Commands modelliert werden. Commands werden von Clients generiert, lokal angewendet und anschliessend an den Server gesendet. Dieser führt den Zustand des Dokuments und wendet Änderungen darauf an. Allfällige Konflikte werden dabei auf dem Server gelöst. Nach der Verarbeitung eines Commands veröffentlicht der Server alle angewendeten Commands sowie Commands zur Konfliktlösung. Diese Commands werden wiederum von den Clients angewendet, um einen konsistenten Zustand des Dokuments herzustellen. Commands werden in einer Datenbank persistiert.

Dieser Ansatz erlaubt es Änderungen nachzuvollziehen und bei Bedarf einen früheren Zustand des Dokuments wiederherzustellen. Das Führen des Zustands auf dem Server erlaubt es, eine ”Source Of Truth” zu haben, die den gültigen Zustand des Dokuments diktiert. Das Anwenden von Commands in den Clients erlaubt es auch dort einzelne Änderungen nachzuvollziehen und darzustellen.

2.2 Technologien und Systemaufbau

Das Versenden von Commands findet über zwei getrennte Kanäle statt. Für Versenden von Commands vom Backend an Clients verwenden wir HTTP-Event Streams (Server Sent Events). Als konkrete Implementation dieser Technologie setzen wir Spring-WebFlux ein. Die weitere Technologieauswahl orientiert sich an diesem Grundsatz Entscheid. Für das Versenden von Commands von Clients an das Backend werden HTTP-POST-Requests verwendet. Diese Trennung erlaubt es, das Schreiben und Lesen von Änderungen zu trennen.

2.2.1 Backend Server

Für das Backend wird eine Spring Boot Applikation mit Spring Webflux erstellt. Dieser Ansatz ermöglicht es, eine reaktive Serverapplikation zu erstellen, welche mit minimalem Boilerplate Code auskommt.

Das Spring Boot Ökosystem benötigt 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 oder reaktiver Programmierung. Wir möchten diese Gelegenheit nutzen, die Sprache in einem Projekt näher kennenzulernen. Wir erhoffen uns vom Einsatz von Kotlin folgende Vorteile:

- Robuste Implementierung dank Null Safety
- Weniger Boilerplate und damit übersichtlichere Implementeirung
- Effiziente und Übersichtliche Anwendung von Streams

2.2.2 Frontend Clients

Für die Client-Applikation wird das Framework React eingesetzt. React bietet mit seinem Komponenten-Model eine einfache Abstraktionsmöglichkeit um die Anwendung sauber zu Kapseln. Die Funktionalen React Komponenten scheinen leichtgewichtiger im Vergleich zu den HTML-Template-Ansätzen von Angular oder VueJS. Das Projektteam hat bis jetzt noch keine grösseren Projekte mit React umgesetzt. Unser Ziel ist es in diesem Projekt die React 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.

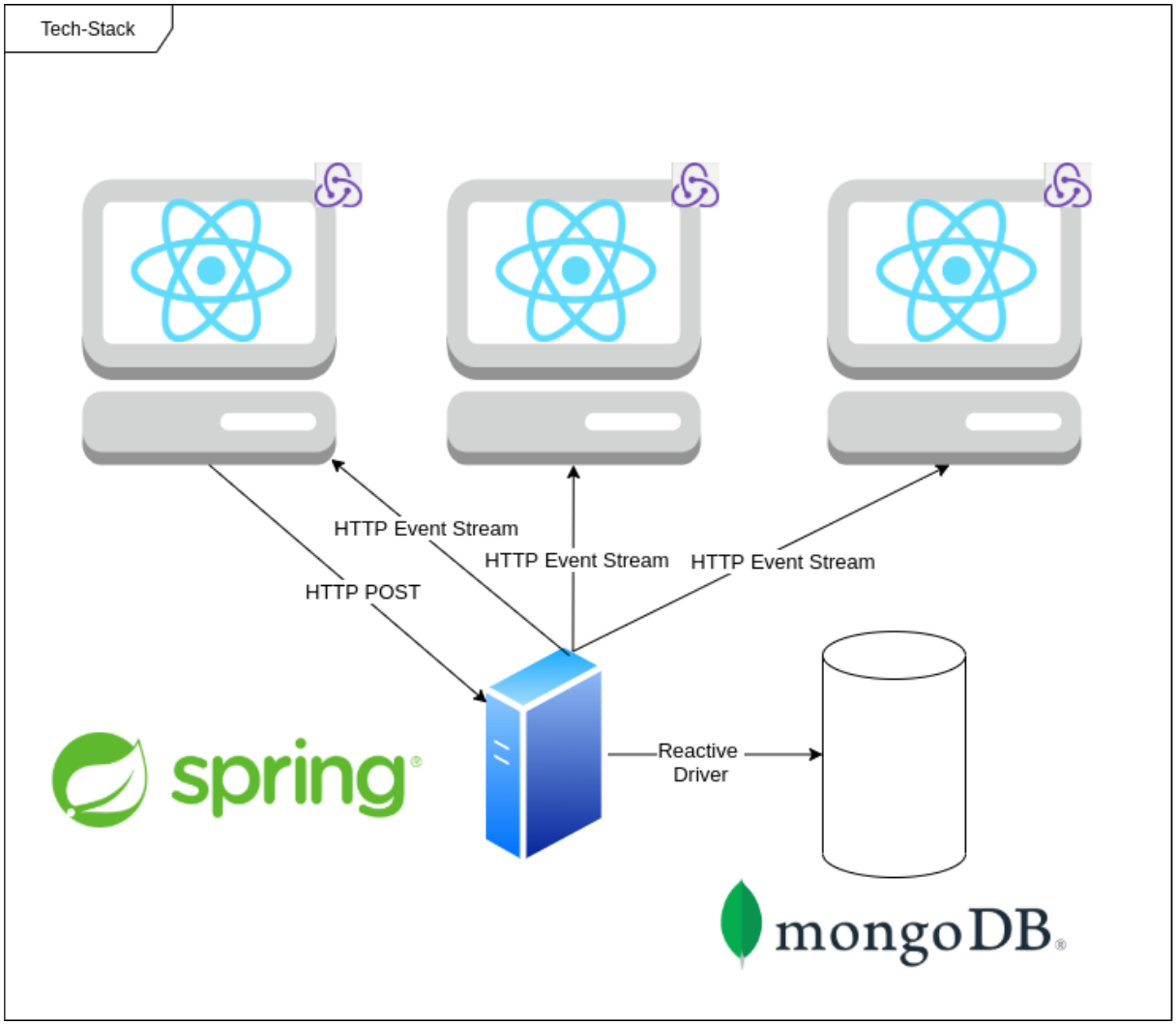

Abbildung 2.1: Technologie Stack

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

2.3 Applikationsprotokoll

In der Folge wird auf die verschiedenen Kanäle und das Applikationsprotokoll eingegangen.

2.3.1 EventSource

Für die stetige Verbindung der Clients zum Server abonnieren die Clients eine EventSource. Die EventSource ist eine langlebige HTTP Verbindung. Über diese Verbindung hören alle Clients auf Änderungen, die von anderen Clients gemacht werden. Dieser Kanal sendet alle Commands einzeln an die Clients. Im Fehlerfall wird die Verbindung geschlossen. Clients können die Verbindung darauf erneut, öffnen um den aktuellen Zustand des Dokuments zu erhalten. Damit wird ein konsistenter Zustand erstellt. Die EventSource wird auch verwendet, um eine Aussage über die verbundenen Clients machen zu können.

2.3.2 HTTP

Die Clients wenden im Frontend alle Commands auf sich selber an und senden danach das Command an den Server. Dafür gibt es einen HTTP POST Endpunkt, der alle Commands entgegennimmt. Intern wird anhand des typeïn der Payload entschieden, wie mit dem Command umgegangen wird. Die genaue Übersicht der Endpunkte ist im Kapitel ÄPIßu finden. Eine Beschreibung der implementierten Typen ist im Kapitel 5 beschrieben.

2.3.3 Applikationsprotokoll

Frontend und Backend tauschen Commands im JSON-Format aus. Dabei haben die ausgetauschten Objekte folgenden Aufbau:

- type: Der Command Type (Bsp. ADD_PARAGRAPH)
- payload: Ein JSON Objekt welches die Payload für den konkreten type enthält
- sender: Eine eindeutige ID des Senders
- (opt.) correlationId: ID der payload eines anderen Commands

2.4 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.

3 Frontend

Das Fronted 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

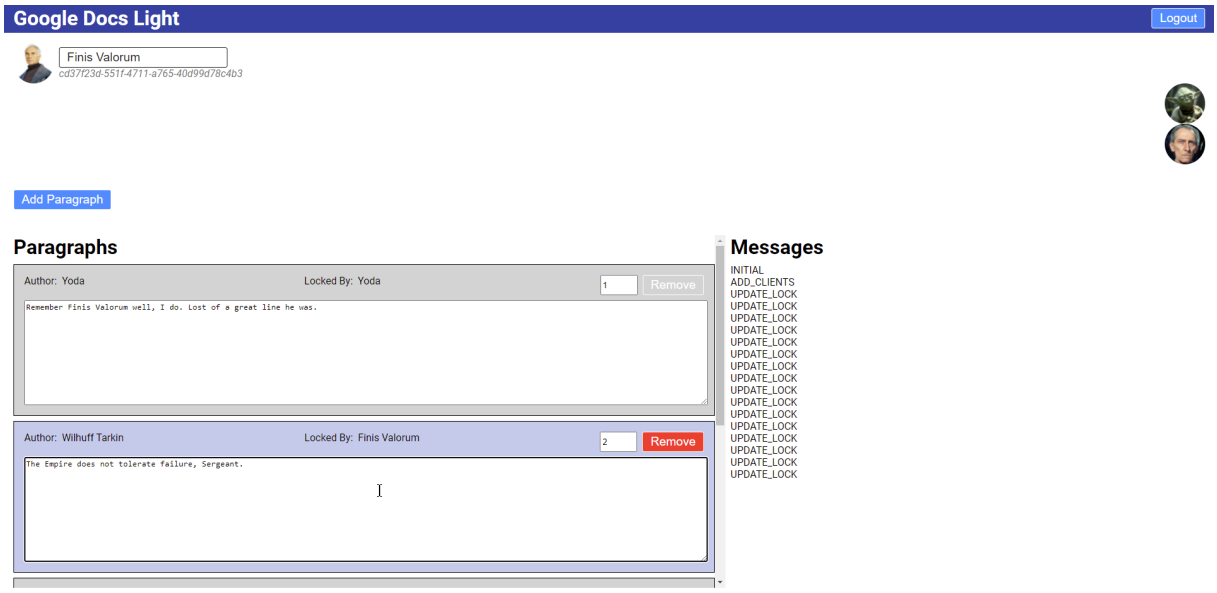

Abbildung 3.1: Team Document User Interface

3.1 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.

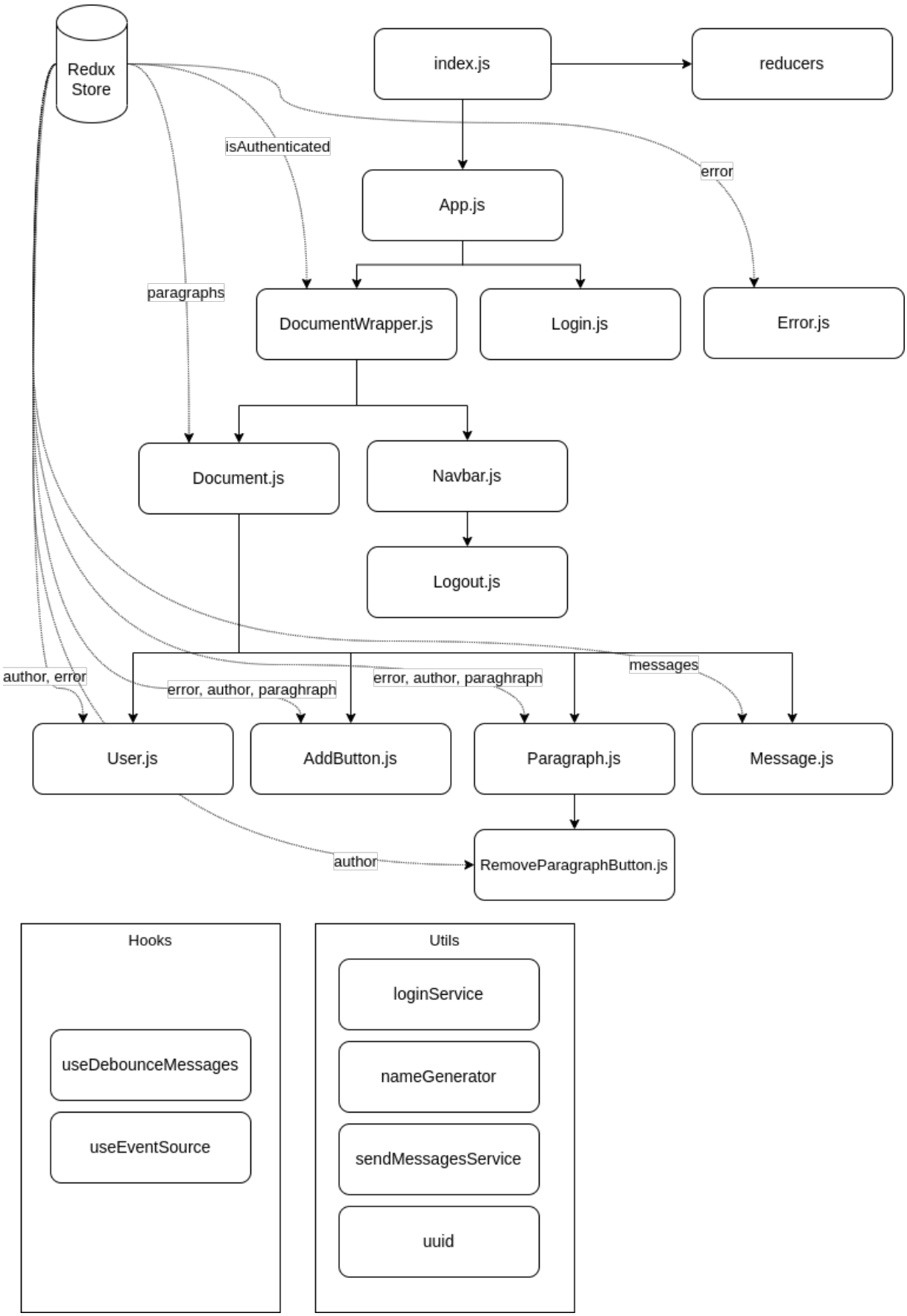

Abbildung 3.2: Komponenten Struktur

3.1.1 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 wir, 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.

3.1.2 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.

3.1.3 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 Eventsource. 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

[2]

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. Dadruch 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 einzeln 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 signifikaten Entlastung der Netzwerklast geführt.

[2] `https://github.com/Yaffle/EventSource#browser-support`

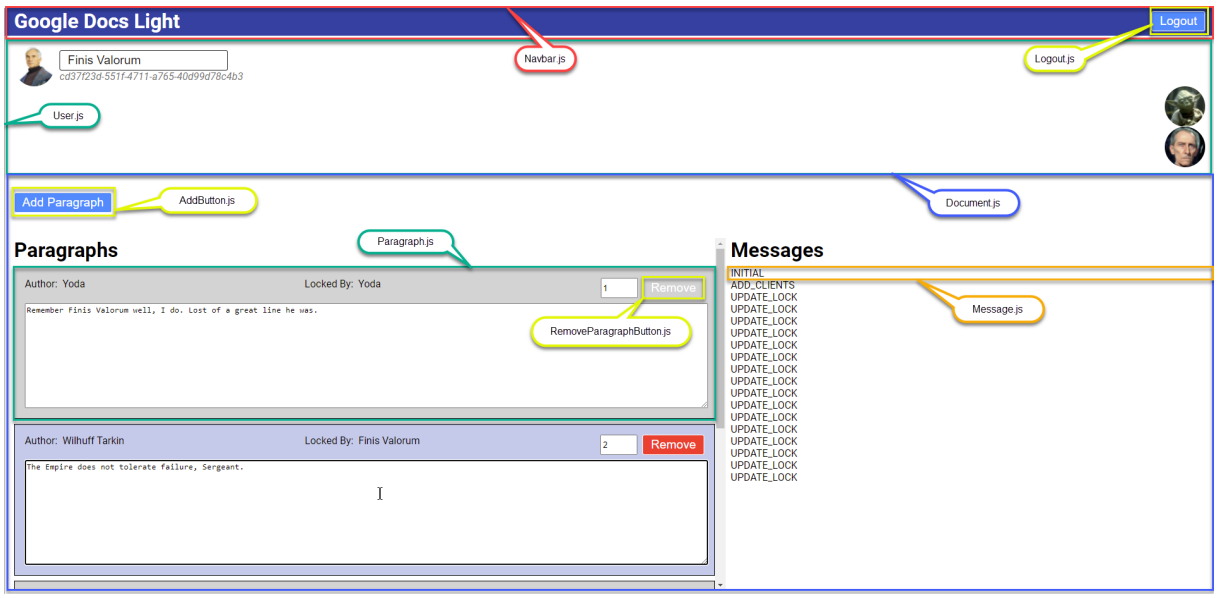

Abbildung 3.3: UI-Components

3.2 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:	ObjectisPresent: 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 korrekturen werden dann von allen Clients verarbeitet.

Das nachfolgende Ablaufdiagramm stellt die möglichen Varianten von Mutationen innerhalb des Frontends dar.

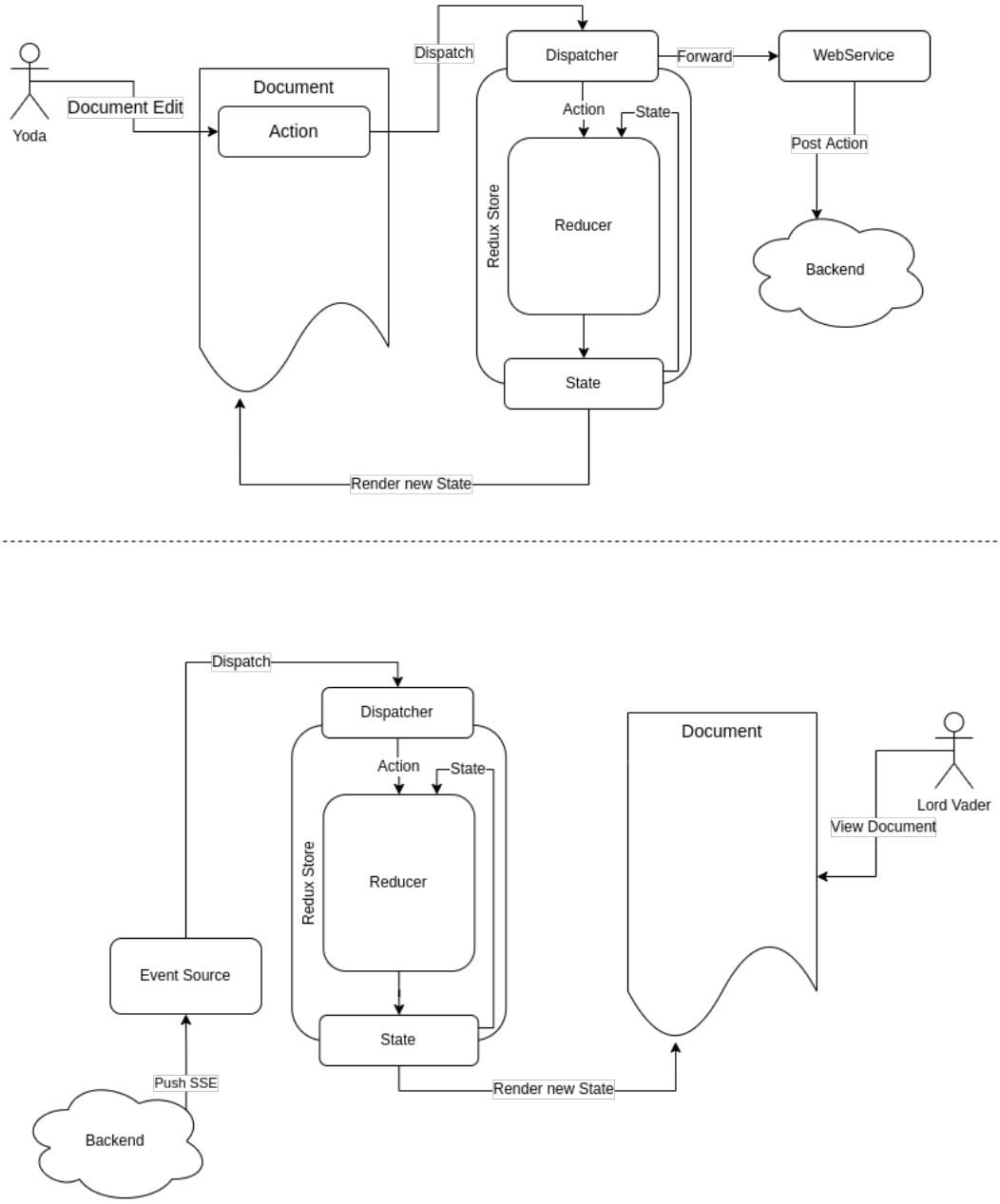

Abbildung 3.4: Ablaufdiagramm

3.3 Fehler Behandlung

Fachliche Fehler werden vom Client nicht behandelt. Möglich 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.

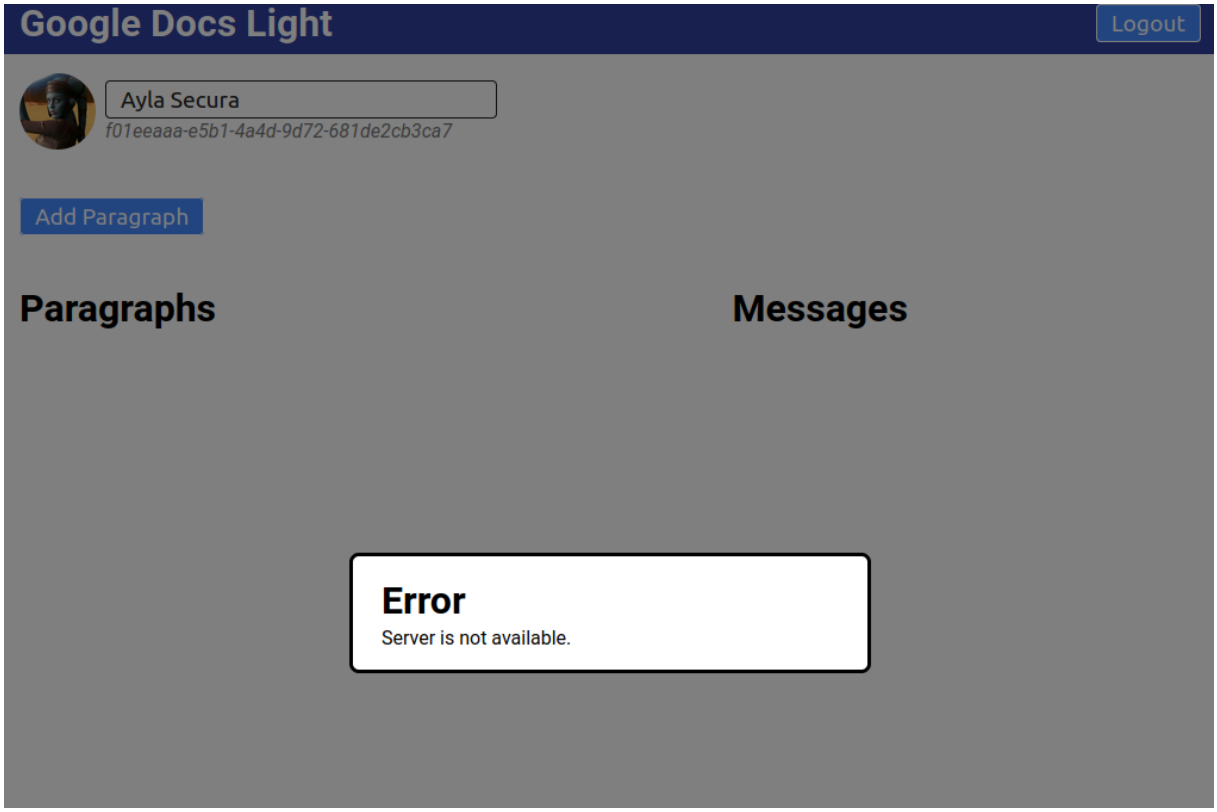

Abbildung 3.5: Fehler Meldung

Beim wiederherstellen der Verbindung zum Server wird von diesem der aktuelle Zustand des Dokumentes mit einem Initial-Command geladen.

4 Backend

4.1 Aufbau

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

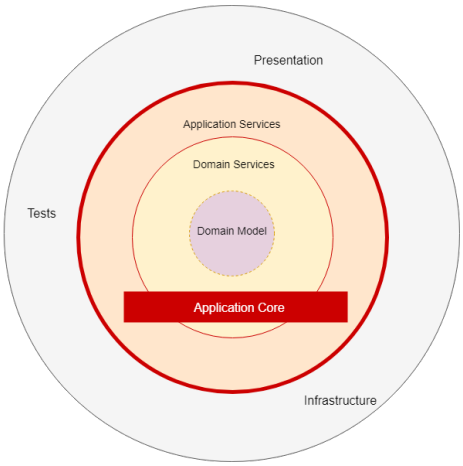

Abbildung 4.1: Onion Architecture

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 Packagestruktur 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 deshalb 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 4.2: Packagestruktur Serverapplikation

Der Domain Layer wird durch das Package domain abgebildet. Dieses beinhaltet die Domänenobjekte und darf keine Abhängigkeiten auf andere Module 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 wird mit den Packages persistence und web 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.

4.2 API

Die Backendapplikation bietet eine HTTPSchnittstelle, 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”.

4.2.1 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

4.2.2 API Document

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

4.2.3 API Message

Beschreibung:	Änderung an Dokument vornehmen
Endpunkt:	/api/v1/message
Methode	POST
Headers:	Authentication: Basic
	Content-Type: application/json
Body:	DocumentCommand als application/json
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:	204, 401 oder 500

4.3 Komponenten

4.3.1 Package Domain

Abbildung 5.3 zeigt die Klassen, des Packages Domain. Sämtliche Klassen in diesem Package besitzen einen öffentlichen Konstruktor, welcher für alle Instanzvariablen einen Parameter entgegennimmt. Diese Konstruktoren sind in der Abbildung nicht abgebildet. Im Zentrum der Domäne stehen die beiden Klassen DocumentCommand und Document.

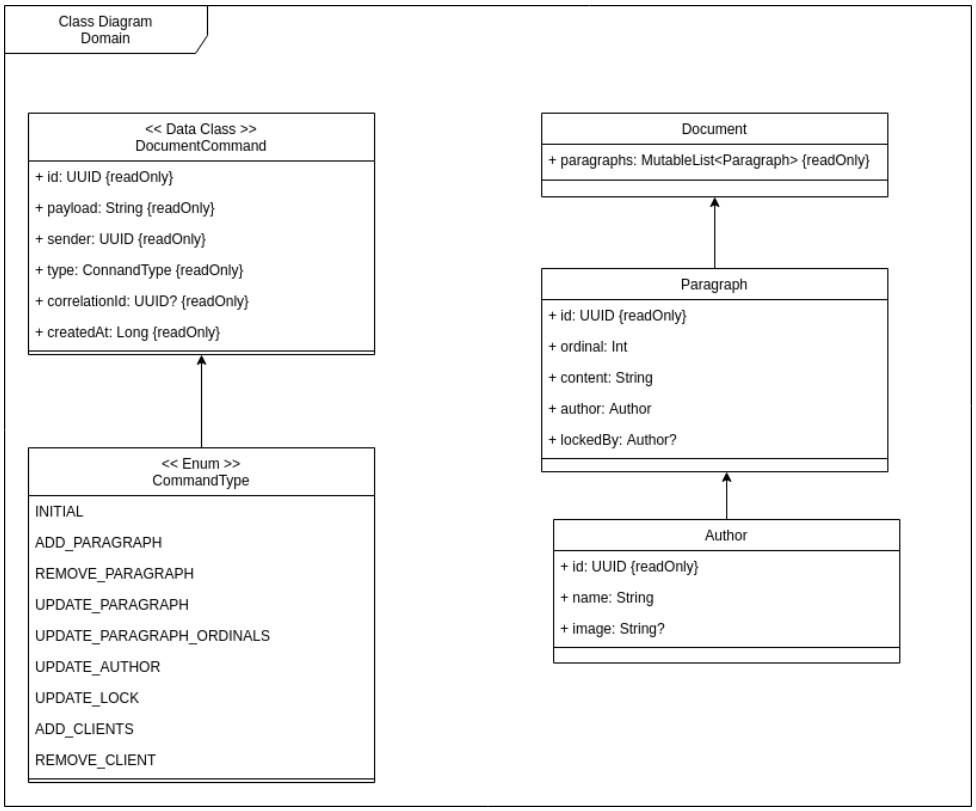

Abbildung 4.3: Klassendiagramm Domain

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 und ist einem Author zugewiesen. 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. 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 DocumentInstanz dar. DocumentCommands werden als einzige Entität persistiert. Damit DocumentCommands eindeutig identifiziert werden können, beinhaltet jede Instanz ein Attribut ”id” vom Tye 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 JSONserialisierte Daten, welche die vorzunehmenden Änderung darstellen. Das Feld ”type” beinhaltet einen Wert aus der Enum CommandType. Dieser Wert kann in den Serviceklassen verwendet werden, um die Payload korrekt zu deserialisieren und die nötigen Änderungen am Dokument 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 Identifkation der Payload eines Commands zu verwenden, ohne die Payload 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. Der verwendete CommandType bestimmt, wie ein Command verarbeitet wird. Dies ist in Kapitel 5.5 weiter beschrieben.

4.3.2 Package Web

AuthenticationController

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

CommandController

Die Klasse CommandController implementiert einen Spring RestController. Dieser Controller stellt zwei POSTEndpunkte zur Verfügung. Über den ersten Endpoint kann eine Liste von DocumentCommands an den Server gesendet werden. Der Endpunkt 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 Endpunkt kann ein gelöschter Paragraph wiederhergestellt werden. Die entsprechende Fachlogik wird an den DocumentSerivce delegiert. Letztlich stellt der Controller einen DELETEEndpunkt zur Verfügung. Über diesen kann der Zustand des Dokuments zurückgesetzt werden.

DocumentStreamUpdateController

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

4.3.3 Package Services

Abbildung 4.4: Klassendiagramm Services

DocumentService

Die Klasse DocumentService ist dafür verantwortlich, erhaltene Anfragen für Dokumente und Änderungen an Dokumenten zu verarbeiten. Sie delegiert die entsprechende Fachlogik an die 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 privates Attribut document vom gleichnamigen Typ. Über die Methode process kann ein einzelner DocumentCommand angewendet werden. Der Processor verarbeitet den 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. Der Service bietet Methoden um die aktiven Benutzer auszulesen, einen neuen Benutzer zu registrieren und einen Benutzer zu entfernen.

4.3.4 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.

4.3.5 Konfiguration

Spring

Alle Klassen im Package SServiceßind 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 "webßind 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 in den Serviceklassen verwendet werden, um auf Werte aus dem application.yaml zuzugreifen.

WebConfig

Die Klasse WebConfig beinhaltet die Konfiguration für SpringSecurity.

4.4 Abläufe

4.4.1 Dokument laden und Änderungen abonnieren

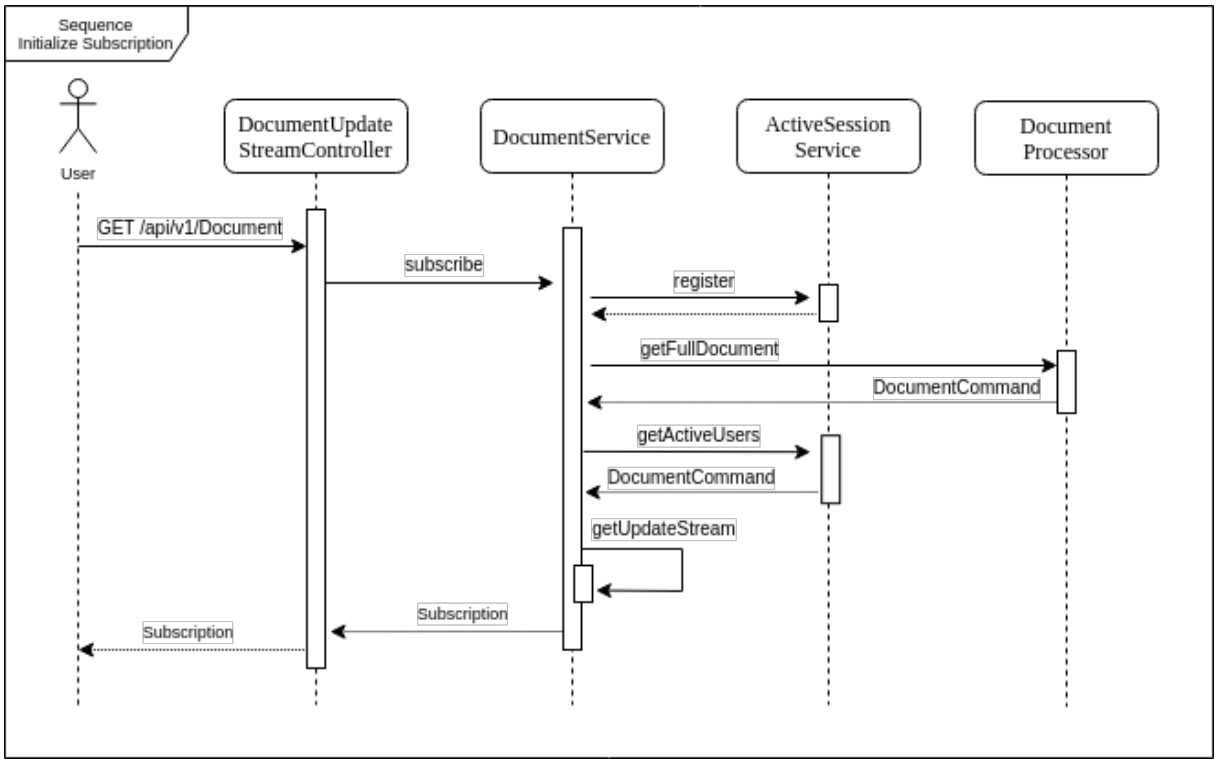

Abbildung 4.5: Sequenzdiagramm Document Subscription

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 in einem einzelnen Flux zusammengefasst und zurückgegeben. Nachdem der Flux geschlossen wird, wird die Registrierung des aktiven Benutzers im ActiveSessionService wieder entfernt.

4.4.2 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 Typ, 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 DokumentCommands 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.

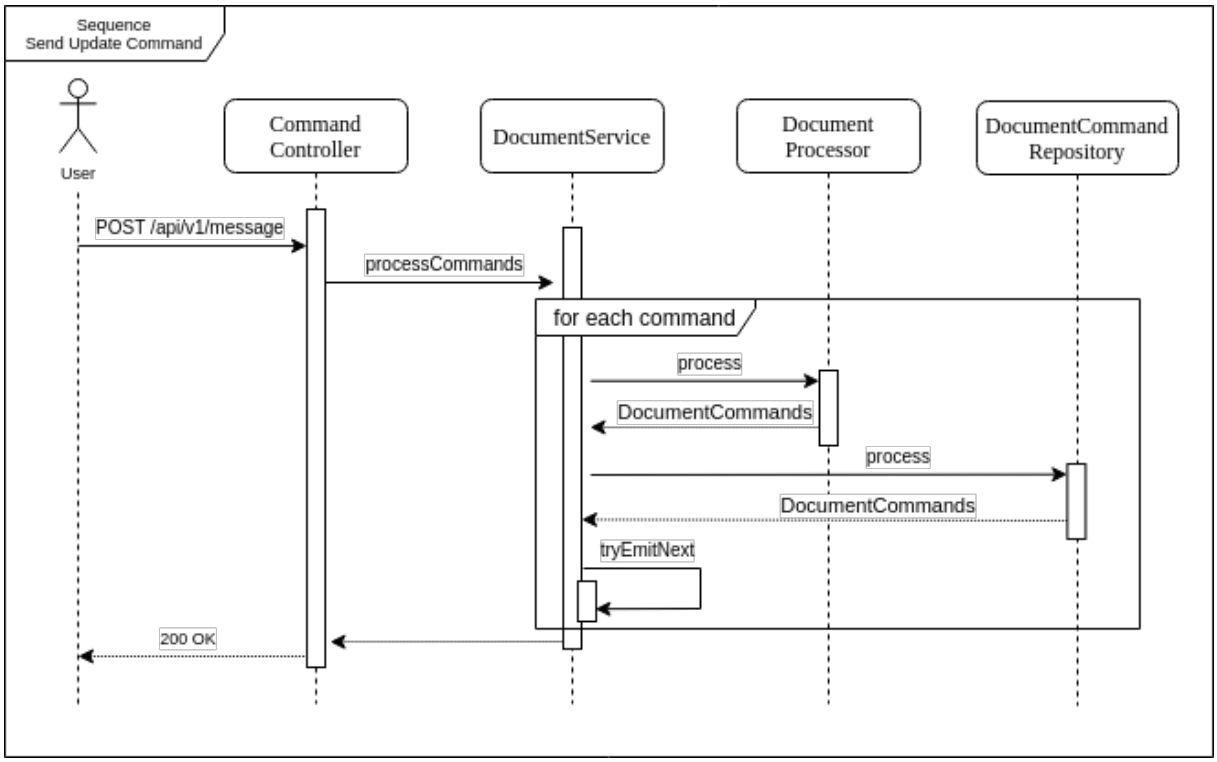

Abbildung 4.6: Sequenzdiagramm Command verarbeiten

4.4.3 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 die Exception in das Log geschrieben und 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.

4.5 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.

4.5.1 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(mutableListOf¡Paragraph¿())” 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.

4.5.2 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 die ADD_PARAGRAPH und UPDATE_PARAGRAPH_ORDINALS Commands zurückgegeben, damit Sie veröffentlicht werden können.

Ein DocumentCommand mit Typ **REMOVE_PARAGRAPH** entfernt einen Paragraphen aus dem Dokument. Ein 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 Pargraphen 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 erhaltenen Command und 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 Pargraphen, der gesperrt werden soll. Dieser Paragraph wird in der Liste von Pargraphen gefunden und durch setzten 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 bei der Registrierung einer Subscription erstellt und veröffentlicht.

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. Der Command wird durch den ActiveSessionService generiert und vom DocumentProcessor verarbeitet, nachdem ein Client die Verbindung getrennt hat.

5 Testing

Das Testing der Applikation soll einerseits Unit Tests in Frontend und Backend beinhalten, sowie Integration Tests, die das Verhalten der kompletten Applikation überprüfen. Eine der identifizierten Problemquellen ist das Netzwerk. Dieses wird aber nicht explizit weiter getestet oder simuliert, da es kein Teil der Nutzererfahrung ist, die wir beeinflussen können. Für die einzelnen Teile der Applikation sollen Testframeworks angewandt werden, die grosse Verwendung finden und für die ausgewählte Technologie üblich ist.

5.1 Frontend

Für das Frontend wird Jest[3] verwendet.

Jest folgende Vorteile:

1. Integration für React und Babel
2. Einfachheit
3. Ohne Browser ausführbar
4. Einfach aufzusetzen
5. Schnell in der Ausführung und dem Feedback

Es sollen einzelne Komponenten getestet werden, damit deren Funktionalität gegeben ist. Es wird darauf verzichtet Enzyme als zusätzliche Dependency einzubinden, da die Tests einfach gehalten werden sollen. Da die einzelnen Komponenten oftmals nur Commands an Redux dispatchen, ist deren Testing als abgeschottete Komponente nicht immer zielführend, da indirekt die Funktionalität von Redux getestet wird. Es bietet sich deshalb an, mehrere Funktionen in die End to End Tests zu verschieben.

5.2 Backend

5.2.1 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.

5.2.2 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.

5.2.3 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.

[3] https://jestjs.io/https://jestjs.io/

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.

5.3 End to End Test

Für das End to End Testing wird Cypress verwendet.

Cypress bietet folgende interessanten Features:

1. Simulation eines Browsers
2. DOM Traversierung anhand von CSS Selektoren
3. Überprüfen von Attributen, CSS Klassen, Value, ...
4. Aufnehmen von Test Cases mittels experimentellen Features im Cypress Studio

Cypress bietet aber explizit ein wichtiges Feature nicht: Es ist nicht möglich mehrere Tabs oder Browser Fenster zu simulieren. Aus der offiziellen Dokumentation: [4]: There will never be support for multiple browser tabs."

Konkret bedeutet dies für das Testing der Applikation, dass nicht mehrere Nutzer mit Cypress simuliert werden können. Um dem entgegenzuwirken, wird eine Klasse für einen Dummy User geschrieben, der direkt die API der Applikation ansteuert. Mehrere Instanzen dieser Klasse können somit die Multiuser Interaktion simulieren. Damit kann getestet werden, dass der maschinell gesteuerte Anwender von Cypress einen konsistenten Zustand beim Arbeiten antrifft. Auch für andere Basis Funktionen wie das Locking ist es notwendig mehrere Nutzer einzubringen.

Da die End to End Test auf die Datenbank zugreifen, ist es ebenfalls notwendig einen Endpunkt anzubieten, der den Zustand des Dokuments zwischen den Ausführungen zurücksetzt.

5.3.1 Probleme

Nebst der angesprochenen Einzelnutzer Limitierung gibt es noch andere Herausforderungen im Zusammenhang mit dem End to End Testing. Da jedes Update ein eigenes Command erzeugt, kann es es sein, dass Cypress beispielsweise zu schnell tippt und bereits die nächste Aktion ausführt, bevor das Backend die Inputs verarbeiten konnte. Beim normalen Arbeiten mit dem Dokument würde einfach der letzte konsistente Zustand hergestellt. Beim Test ist es aber notwendig, dass von einem Bestimmten Output oder Status in der Applikation ausgegangen werden kann, um die Testbedingung zu erfüllen. Damit dies beim Testen nicht zum Problem wird, werden öfters cy.wait() Statements und delay Optionen auf Input aufgerufen. Dies ist aus Sicht des Testings vertretbar, da ein Mensch auch einen Abstand zwischen einzelnen Tastenanschlägen hat, eine Think Time"hat oder auch auf das visuelle Feedback der Applikation warten kann. Einzelne cy.wait() Statements werden beispielsweise verwendet, um das Laden der Applikation nach dem Login oder das Zurücksetzen des Dokuments abzuwarten.

5.3.2 Stress Tests

Die Stress Tests werden mit mehreren Instanzen der User Klasse ausgeführt. Der Üser"verwaltet dabei einen eigenen State seiner Paragraphen in Memory und ruft die API auf, um seine eigenen Paragraphen im Dokument zu verändern. Der Cypress User interagiert mit diesen anderen Usern und versucht dabei seine Aufgaben abzuarbeiten.

[4] https://docs.cypress.io/guides/references/trade-offs

Stress Tests auf dem Live System wurden unsystematisch während der Entwicklung durch die Entwickler mit mehreren Browsern ausgeführt. Dadurch konnten jedoch verschiedene Probleme festgestellt werden, wie zum Beispiel die Limitierung über das Netzwerk oder die Notwendigkeit eines grösseren Buffers.

6 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.

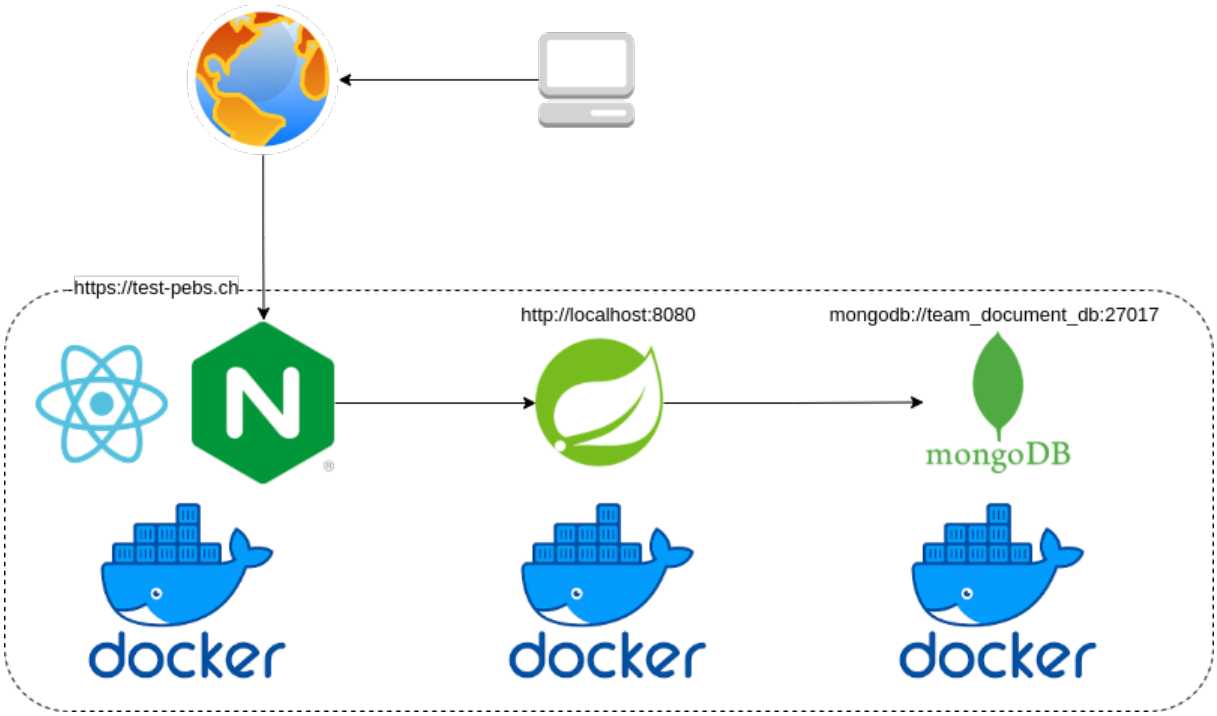

Abbildung 6.1: Deployment Übersicht

7 Fazit

Die umgesetzte Lösung erlaubt es ein Dokument parallel auf einem öffentlich erreichbaren Server zu bearbeiten. Die funktionalen Anforderungen an die Applikation sind grösstenteils umgesetzt. Das System hat aber noch diverse Lücken in den Bereichen Konfliktlösung und Zustandsverwaltung. In gewissen Situationen ist es möglich, dass inkonsistente Ordinalnummern entstehen und Änderungen verloren gehen.

Der Ansatz im Backend mit einem reaktiven Stack zu arbeiten und Updates über die Eventsource zu versenden funktioniert sehr gut. Neue Funktionalität könnte elegant mit zusätzlichen Command-Types hinzugefügt und in die Processor Chain eingebunden werden. Der Processor erreichte im isolierten Lasttest eine durchschnittliche Verarbeitungszeit von 4 ms für einen einzelnen Command. Der Application Test von Controller bis zur Eventsource brauchte im Schnitt 40 ms für die komplette Verarbeitung.

Analog scheint auch der Ansatz im Frontend mit React und Redux gut zu funktionieren. Die Verarbeitung der Updates von der Eventsource lässt sich mit dem gleichen Prinzip wie im Backend einfach erweitern. Ein neuer Command entspräche dann einer neuen Funktion im Reducer. Die Redux Komponenten kommen ohne eigene Logik aus, was die gesamte Frontendapplikation übersichtlich und einfach zu warten macht. Die Implementation der Reducer-Funktionen hat sich als herausfordernder als erwartet dargestellt. So ist es im Endeffekt nicht gelungen, Reducer zu implementieren, welche einen garantiert konsistenten Zustand herstellen.

Was sich retrospektiv als Fehler erwiesen hat, war die Entscheidung die Commands aus dem Frontend via HTTP an den Server zu senden. Für jedes Update wird eine TCP Verbindung aufgebaut und entsprechend auch gewartet bis diese wieder abgebaut ist. Das führt bei schnellen aufeinanderfolgenden Updates über das Internet zu einem Request-Stau. Treffen am Server mehr Requests ein als dieser verarbeiten kann ist auch die Reihenfolge der eingehenden Commands nicht mehr garantiert. Diese Probleme wurden erst nach erfolgreichem Deployment auf dem öffentlichen Server sichtbar. Aufgrund des schon fortgeschrittenen Projektstandes haben wir dieses Grundkonzept nicht gänzlich überarbeitet, sondern versucht das Netzwerk mit einer debounce funktion zu entlasten. Die Konfliktbehandlung des Backends geht davon aus, dass Updates in der korrekten Reihenfolge (FIFO) ankommen. Da es dennoch noch vorkommen kann, dass sich einzelne Requests überholen und die eigenen Updates vom Client direkt im Redux Store angewendet werden, können inkonsistente Zustände im Frontend entstehen. Dieses Problem ist die zweite Ursache für die bekannten Lücken im Bereich Konfliktlösung und Zustandsverwaltung.

Am Ende dieses Projektes würden wir die Architektur des Systems grösstenteils beibehalten. Dabei allerdings das Versenden von Commands von Backen und Clients immer über denselben Kanal gehen lassen. Die Implementierung der Reducer müsste überarbeitet werden, um alle Konflikte korrekt behandeln zu können.

Abbildungsverzeichnis