Collaborative Markdown Editor

Dokumentation zur Webanwendung
eines kollaborativen Markdown-Editors

in der Vorlesung Java EE

von Morten Terhart, Micha Spahr und Michael Skrzypietz

Duale Hochschule Baden-Württemberg Mosbach
Kurs INF16B

11.06.2019

Vorlesung: Java EE
Dozent: Herr Dr. Alexander Auch
Kurs: INF16B

Inhaltsverzeichnis

1 Projektidee

Unsere Idee entstand bereits im 4. Semester. Zur Klausurvorbereitung fasste insbesondere Morten einige Vorlesungsskripte zusammen und hat hierfür den Online-Markdown-Editor StackEdit verwendet. Markdown ist eine Markup-Sprache, um Textdokumente mit wenig Syntax strukturieren und formatieren zu können. Das große Problem mit StackEdit war jedoch, dass es nicht mehrbenutzerfähig ist.

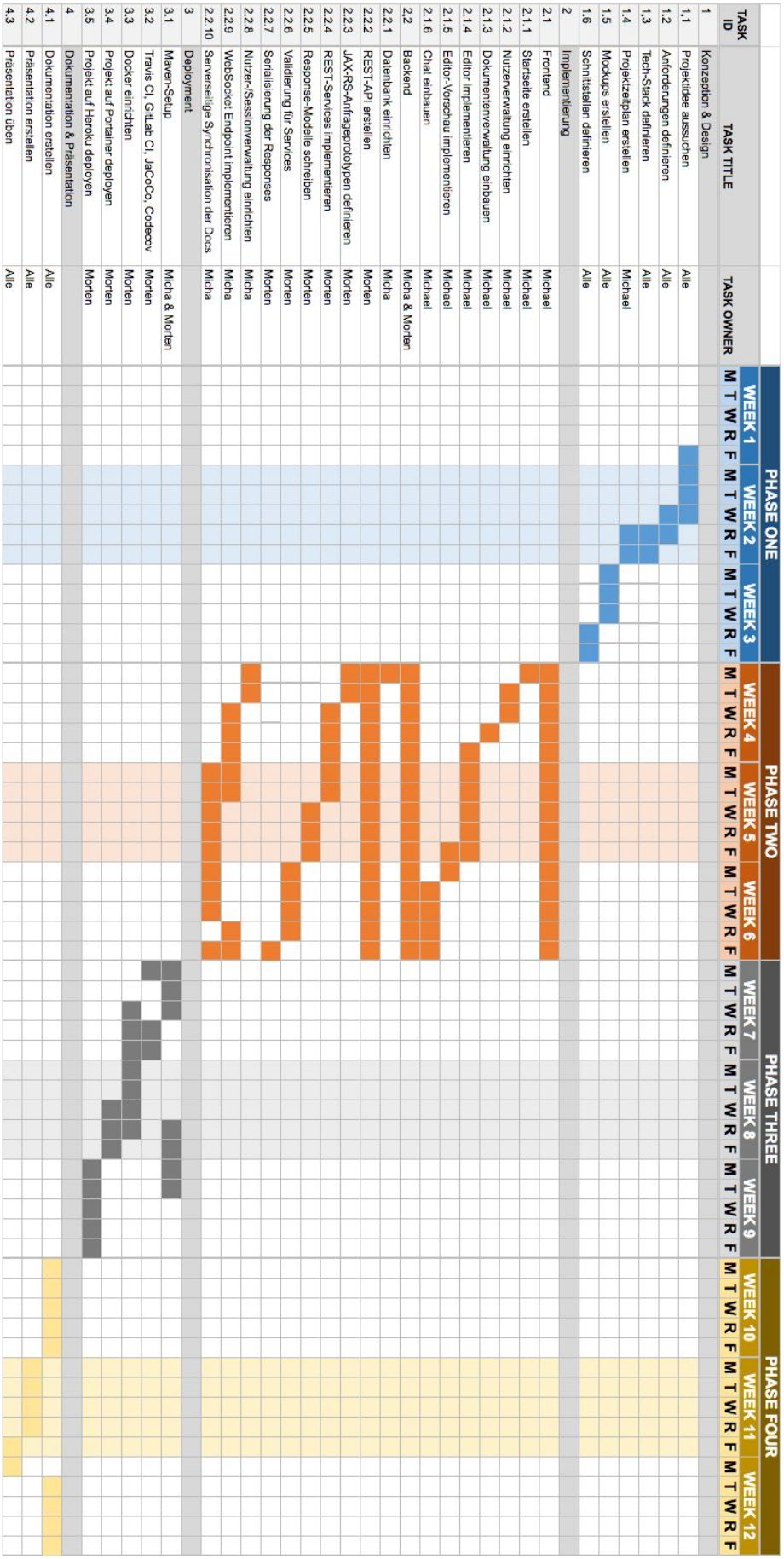
Zwar haben wir nach erneuter Recherche auch kollaborative Markdown-Editoren, wie z. B. https://hackmd.io, gefunden, jedoch fanden wir die Herausforderung der Synchronisation der einzelnen Clients so spannend, dass wir das Projekt dennoch angehen wollten. Zudem wollten wir die WebSockets praktisch kennen lernen.

Folglich sind unsere Anforderungen an unseren Editor:

- Nutzerverwaltung
- Erstellung von neuen Dokumenten im Workspace von Nutzern
- Löschen von Dokumenten
- Synchronisation des Editors zwischen mehreren Clients
- Unterstützung der Markdown Syntax
- WYSIWYG Toolleiste für Markdown
- Vorschau der Änderungen in Echtzeit
- Hinzufügen und Entfernen von Mitarbeitern zu einem Projekt
- Möglichkeit der Übertragung eines Projektes zu einem anderen Nutzer
- Dokumenten-Chat für die Kommunikation zwischen den Nutzern

2 Projektplan

Die nebenstehende Abbildung ist unser Projektplan, welcher die Phasen *Konzeption und Design*, *Implementierung*, *Deployment* und *Dokumentation und Präsentation* umfasst. Die bearbeitenden Personen der Aufgaben sind jeweils neben den eigentlichen Aufgaben aufgelistet sowie eine Kennzeichnung der Wochen und Tage, an denen diese Aufgaben bearbeitet wurden. Die zeitliche Bearbeitung liegt hierbei in Form eines Gantt-Diagramms vor.

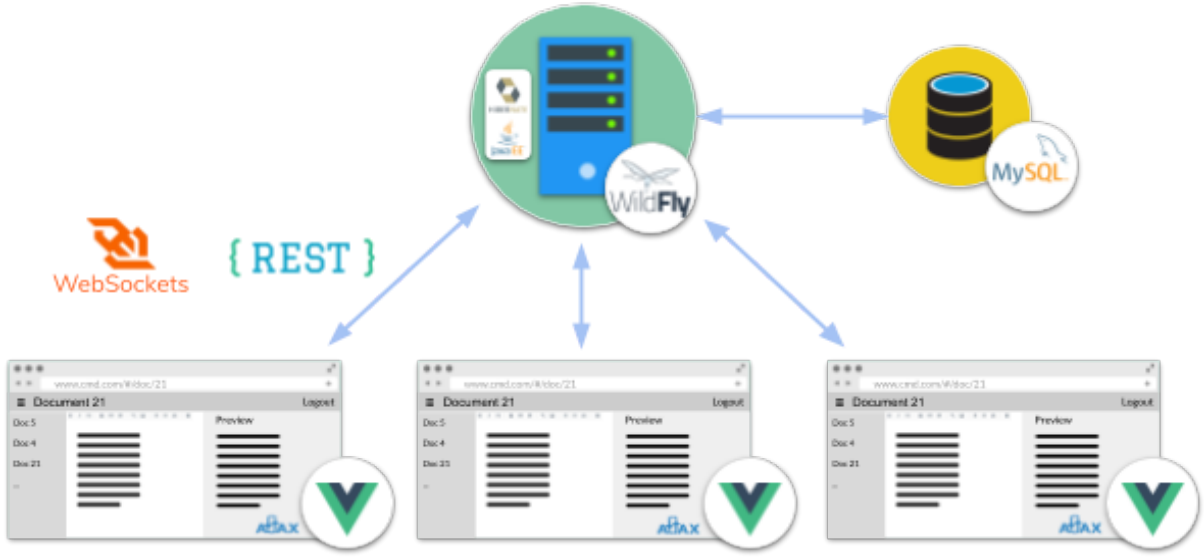


3 Design

Um einen funktionsfähigen Markdown-Editor entwickeln zu können, muss zunächst das Konzept und ein entsprechendes Design dazu erstellt werden. Es wird ein zentraler Webserver benötigt, welcher den Markdown-Editor und die Funktionalitäten zur Zusammenarbeit an Dokumenten bereitstellt. Außerdem wird eine Datenbank benötigt, um die Benutzer, Kollaboratoren und Dokumente sicher zu speichern.

3.1 Architektur

Die Anwendung ist als typische Client-Server-Architektur aufgebaut und besteht aus den Komponenten des Webservers, der Datenbank und der Präsentationsansicht in den Webbrowsern der einzelnen Clients. Der Webserver ist ein WildFly Application Server, welcher den kompletten Java-EE-Stack ausführt. Dazu gehört beispielsweise die JPA-API mit der konkreten Implementierung Hibernate, über die eine Verbindung zur Datenbank hergestellt wird. Als Datenbank wird MySQL verwendet. Webserver und Datenbank können hierbei auf getrennten Rechnern voneinander laufen.

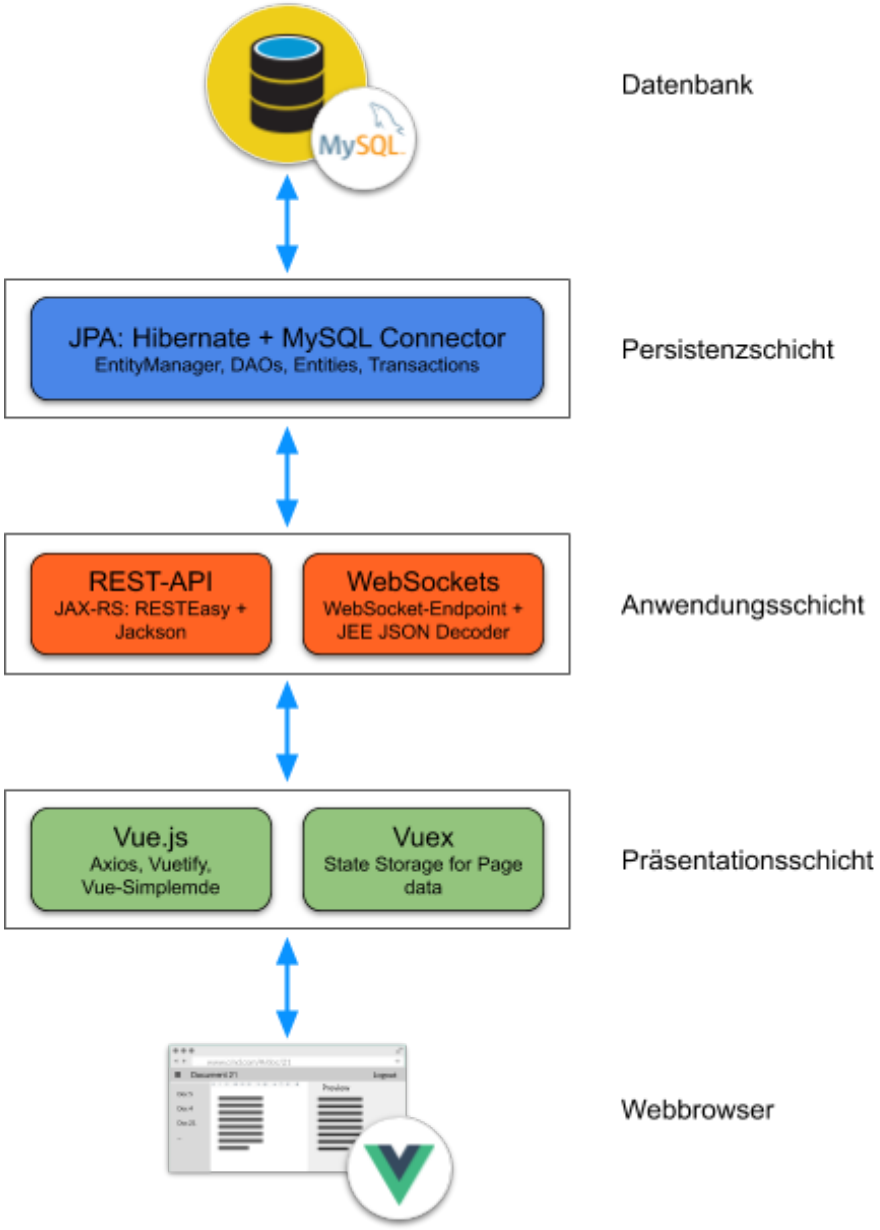


Auf der Client-Seite wird der Webbrowser zur Anzeige des Editors und der Vorschau verwendet. Hierbei wurde das JavaScript-Frontend-Framework Vue.js benutzt. Jeder Client besitzt seine eigene lokale Kopie eines Dokumentes und Änderungen am Dokument werden über das WebSocket-Protokoll mit dem Server ausgetauscht und so an alle anderen Clients geliefert. Grundlegende Funktionalität für die Authentifizierung und Verwaltung von Dokumenten bietet eine REST-API.

3.2 Schichtenarchitektur

Die Anwendung ist in mehreren Schichten aufgebaut, welche unterschiedliche Software und Technologien vereinen. An oberster Stelle steht die Datenbank, die über eine

JDBC-Verbindung mit der Persistenzschicht der Anwendung kommuniziert. Diese basiert auf JPA und macht Datenbankzugriffe mithilfe von Database Access Objects (DAOs). Teilweise sind diese Zugriffe als Transaktionen realisiert. Die Persistenzschicht liefert Daten für die Anwendungsschicht der Software, die aus der REST-API und einem WebSocket-Endpoint besteht. Über diese Schicht können Daten an die Clients in der Präsentationsschicht geliefert werden. So kommuniziert das Frontend mit der REST-API, um bestimmte Daten abzufragen oder Operationen durchzuführen, und nutzt WebSocket-Verbindungen zur Synchronisation der Dokumentenstände. Die Präsentationsschicht besteht aus dem Frontend mit Vue.js und Vuex und generiert die HTML-Repräsentation der Website im Webbrowser des Clients.



4 Installation der Anwendung

Da die Anwendung aus unterschiedlichen Komponenten besteht, müssen diese auch entsprechend richtig eingerichtet werden. Die Anordnung und Funktion der Komponenten wurde bereits in der Architektur betrachtet.

Für den Betrieb des Backends und die Auslieferung des Frontends ist ein JBoss WildFly AS in der Version `16.0.0.Final` vorgesehen. Außerdem muss sichergestellt sein, dass Java 11 auf dem System installiert ist. Als Datenbank wird MySQL verwendet, entweder in der Version `5.7` oder `8.0`.

4.1 Automatische Einrichtung mit Docker-Compose

Für den WildFly Application Server und die MySQL-Datenbank liegen sogenannte Dockerfiles vor, womit Docker Images für diese beiden Komponenten bauen kann. Docker-Compose ist ein Tool, um verschiedene Docker-Container in einem Netzwerk zu verknüpfen und die Konfiguration der Docker-Umgebung zu automatisieren. Obwohl es nicht für Produktionszwecke vorgesehen ist, kann es verwendet werden, um die Anwendung zum Testen zu starten.

Zunächst sollte geprüft werden, welche Versionen von Docker und Docker-Compose vorliegen. Es sollte sich mindestens um Docker Version 18.0 und Docker-Compose 1.23 handeln, damit die deklarierten Features auch vollständig unterstützt werden. Zum Bauen und Starten der Container kann der einfache Befehl

```
$ docker-compose up
```

im Projektverzeichnis benutzt werden. Hier passieren nun eine ganze Reihe an Operationen, die im Folgenden aufgelistet sind.

1. Die Basisimages `mysql` und `jboss/wildfly` werden von Docker Hub heruntergeladen.
2. Das MySQL-Image wird gebaut, ein Root-User und ein Datenbank-User werden eingerichtet und die Datenbank initialisiert.
3. Das WildFly-Image wird gebaut.
4. Zunächst wird die Anwendung mithilfe von Maven in einem anderen Container gebaut. Hier ist der Bau des Frontends direkt mit inbegriffen, da das Maven-Frontend-Plugin verwendet wird.
5. Ein Administrator-Account wird zum WildFly hinzugefügt.
6. Der MySQL-Konnektor wird heruntergeladen und beim WildFly installiert. Außerdem wird direkt eine Datasource eingerichtet, um die Verbindung zur Datenbank im anderen Container zu erlauben.
7. Das vorher durch Maven gebaute WAR-Archiv wird auf den WildFly deployed.

Nach Schritt 7 wird das Docker-Netzwerk zwischen den beiden Containern aufgebaut. Außerdem wird ein Volume angelegt, auf dem die Daten der MySQL-Datenbank gespeichert werden.

Nach dem Bau der Images werden Container erstellt und direkt gestartet, der Log wird entsprechend auf der Konsole ausgegeben. Die Anwendung läuft nun unter der URL http://localhost:8080.

4.2 Maven-Deploy

Alternativ zum Docker-Image kann der Maven Deploy verwendet werden. Dies wird durch das Wildfly Maven Plugin ermöglicht. Auch hier wird durch das Plugin ein Wildfly 16.0.0.Final heruntergeladen, gestartet und mit MySQL-Konnektor und Datasource konfiguriert.

Hierfür muss zunächst der Wildfly mit dem Befehl

```
$ mvn wildfly:start
```

gestartet werden. Dies ist erforderlich, um im Anschluss den MySQL-Konnektor und die Datasource hinzuzufügen sowie die Anwendung zu deployen. Bevor allerdings fortgefahren werden kann, muss sichergestellt werden, dass eine MySQL-Datenbank auf dem Rechner vorhanden ist und läuft. Die Verbindungsinformationen (JDBC-URL, Benutzername, Passwort) müssen in der Maven `pom.xml` in den Properties

- `wildfly.datasource.jdbc.url`
- `wildfly.datasource.username`
- `wildfly.datasource.password`

eingetragen werden, um eine korrekte Verbindung zuzulassen. Wenn dies geschehen ist, kann der Befehl

```
$ mvn deploy -P deployment -DskipTests=true
```

ausgeführt werden, um den kompletten Maven-Lifecycle von Kompilierung der Software, Bauen des Vue.js-Frontends, Generierung eines JaCoCo-Coverage-Reports über das Packen der Anwendung in ein WAR-Archiv bis hin zur Konfiguration des Wildfly und dem Deploy der Anwendung durchzuführen.

Wenn bis hierhin nichts schiefgegangen ist, sollte die Website unter http://localhost:8080 zur Verfügung stehen.

4.3 Manuelle Einrichtung

Neben den automatisierten Einrichtungsverfahren kann der Webserver auch manuell eingerichtet werden. Dazu muss zunächst MySQL installiert werden und Wildfly von der

offiziellen Download-Seite https://wildfly.org/downloads in der Version 16.0.0.Final heruntergeladen werden. Außerdem wird der MySQL-Konnektor in der Version 8.0.15 benötigt, der hier heruntergeladen werden kann.

Zunächst sollte die Datenbank mit dem Befehl

```
$ mysql.server start
```

gestartet werden. Anschließend muss der MySQL-Konnektor beim Wildfly installiert werden. Dies geht am einfachsten mit dem JBoss-CLI, welches sich unter `${JBOSS_HOME}/bin/jboss-cli.sh` befindet. Hierfür muss allerdings der Wildfly-Server laufen, welches man mit dem Befehl

```
$ ${JBOSS_HOME}/bin/standalone.sh
```

rasch erledigen kann. In einem anderen Terminal-Fenster wird dann das JBoss-CLI mit dem Befehl

```
$ ${JBOSS_HOME}/bin/jboss-cli.sh --connect
```

gestartet und folgender Befehl eingegeben (Der Wert in den Klammern <> sollte durch den entsprechenden richtigen Pfad zum MySQL-Konnektor ersetzt werden):

```
module add --name=com.mysql
      --resources=<Pfad zur mysql-connector.jar>
      --dependencies=javax.api,javax.transaction.api
```

Hier wird das Modul für den MySQL-Konnektor hinzugefügt. Um den Treiber nun zu installieren, wird folgendes im JBoss-CLI ausgeführt:

```
/subsystem=datasources/jdbc-driver=mysql:add(
      driver-name=mysql,
      driver-module-name=com.mysql,
      driver-class-name=com.mysql.cj.jdbc.Driver,
      driver-xa-datasource-class-name=com.mysql.cj.jdbc.MysqlX
ADataSource)
```

Nun ist der Treiber als solcher erfolgreich hinzugefügt, anschließend muss die Datasource für die Datenbankverbindung eingerichtet werden. Dies geschieht mit dem folgenden Befehl (Werte in Klammern <> entsprechend durch die richtigen Verbindungsdetails austauschen):

```
data-source add
      --name=cmdDS
      --jndi-name=java:/cmd
      --user-name=<DB Username>
      --password=<DB password>
      --connection-url=jdbc:mysql://localhost:3306/cmd
      --driver-name=mysql
      --max-pool-size=25
      --blocking-timeout-wait-millis=5000
      --enabled=true
      --jta=true
      --use-ccm=false
```

Falls dieser Befehl erfolgreich ausgeführt wurde, ist die Einrichtung des Wildfly abgeschlossen. Das JBoss-CLI kann mit dem Befehl `exit` oder durch Drücken von `<Ctrl-D>` verlassen werden.

Der letzte Schritt ist der eigentliche Deploy der Anwendung. Dafür muss die Anwendung zunächst mit Maven gebaut werden, was mit dem Befehl

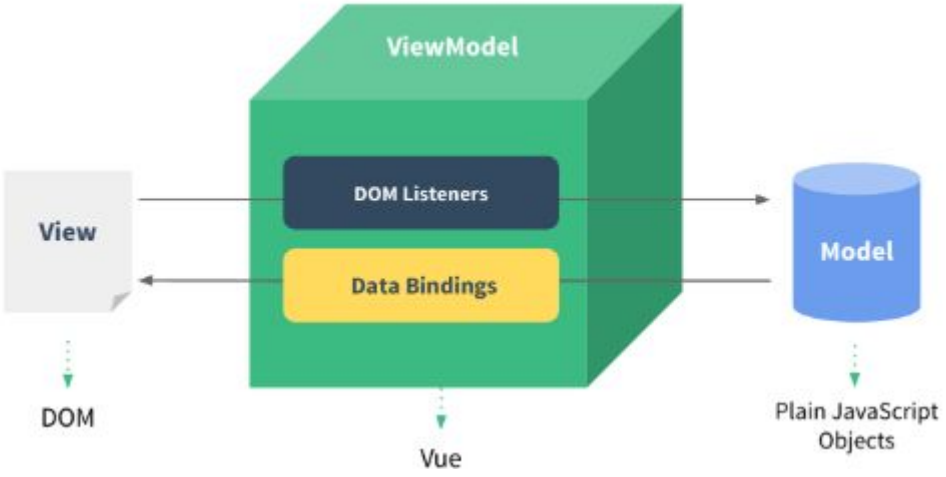
```
$ ./mvnw install -P deployment -DskipTests=true
```

erledigt werden kann. Das daraus erstellte WAR-Archiv befindet sich dann unter `target/ROOT.war` und muss in das Verzeichnis `${JBOSS_HOME}/standalone/deployments` verschoben werden. Zuletzt muss der Wildfly ein letztes Mal neu gestartet werden, um die Anwendung final zu deployen. Das Ergebnis ist dann unter http://localhost:8080 zu sehen, wo die Oberfläche des Markdown-Editors auftauchen sollte.

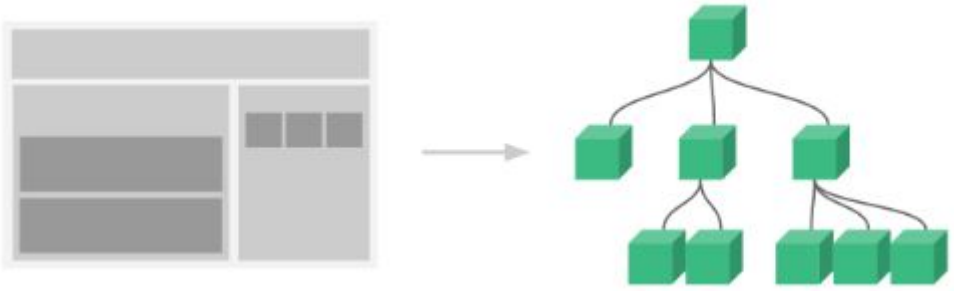
Alternativ zum JBoss-CLI kann auch das JBoss Management Interface zum Konfigurieren des MySQL-Konnektors und der MySQL Datasource verwendet werden, das sich nach dem Start des Wildfly-Servers unter http://localhost:9990 befindet. Hierfür muss allerdings vorher ein Adminstrator-Account beim Wildfly angelegt werden.

5 Frontend

Das Frontend wurde mit der Vue.js-Bibliothek entwickelt und ist somit eine Single-Page-Applikation. Es arbeitet nach dem MVVM-Entwurfsmuster und zeichnet sich durch sein Leichtgewicht aus, da es sich standardmäßig nur um die Präsentationsschicht kümmert. Die großen Vorteile von Vue.js ist die reaktive Datenbindung und die zusammensetzbaren Ansichtskomponenten. Reaktive Datenbindung bedeutet, dass die Daten der Komponenten und ihre Repräsentation im DOM permanent synchron gehalten werden, wodurch man nicht selber den DOM updaten muss.



Das Komponentensystem von Vue.js bietet Entwicklern eine Abstraktion und ermöglicht ihnen sehr große Applikationen aus kleinen, gekapselten und wiederverwendbaren Komponenten zu erstellen. Folglich besteht eine Applikation aus einem Baum von Komponenten.



5.1 Weitere genutzte Bibliotheken

Neben Vue.js wurden noch viele weitere Bibliotheken genutzt (vgl. `./frontend/package.json`), um das Frontend zu erstellen. Wichtige nennenswerte Bibliotheken sind:

- Vuetify bietet gutaussehende, mit Google's Material Design erstellte Komponenten
- Vuex ist ein zentralisierter Speicher des Applikationszustands

- Vue-Router übernimmt die simulierte Anzeige der aktuellen URL
- Axios bietet die Möglichkeit, mit Promises eine REST-API aufzurufen
- Babel ermöglicht die Nutzung der neuen Syntax von ECMASCRIPT 2015+, indem es den Code in reguläres JavaScript kompiliert
- Vue-simplemde ist die genutzte Markdown-Editor-Komponente, welche die Bibliothek marked als Markdown-Renderer benutzt
- Vue-beautiful-chat ist die genutzte Chat-Komponente

5.2 Clientseitige Synchronisation des Editors

Da aus Performancegründen zur Synchronisation des Editors nicht bei jeder Änderung ein eigner WebSocket Event gesendet werden soll, erfolgt die Synchronisation nach einer gewissen Zeit, in der der Anwender keine Änderungen vorgenommen hat.
Beim Laden der Editor-Komponente wird zunächst eine eigene Callback-Funktion zum Change-Event des Editors hinzugefügt, um alle Änderungen eines Dokuments in einer Liste zu speichern. Durch dieses Vorgehen erhält man bereits automatisch eine Liste von Änderungen in der richtigen Reihenfolge.

```
142             let vm = this
143             this.simplemde.codemirror.on("change", function(editor, changeObj) {
144                 if (!changeObj.origin || changeObj.origin === "setValue") {
145                     return
146                 }
147
148                 const deleteMessage = vm.buildMessageStringFromArray(changeObj.removed)
149                 if (deleteMessage.length > 0) {
150                     vm.contentChanges = [...vm.contentChanges, {
151                         type: 'Delete',
152                         msg: deleteMessage,
153                         pos: vm.getTotalCursorPos(vm.content, changeObj.from.line, changeObj.from.ch)
154                     }]
155                 }
156
157                 const insertMessage = vm.buildMessageStringFromArray(changeObj.text)
158                 if (insertMessage.length > 0) {
159                     vm.contentChanges = [...vm.contentChanges, {
160                         type: 'Insert',
161                         msg: insertMessage,
162                         pos: vm.getTotalCursorPos(vm.content, changeObj.from.line, changeObj.from.ch)
163                     }]
164                 }
165
166                 vm.sendContentDiffAfterTyping(vm)
167                 vm.$emit('contentWasChanged', vm.content);
168             })
```

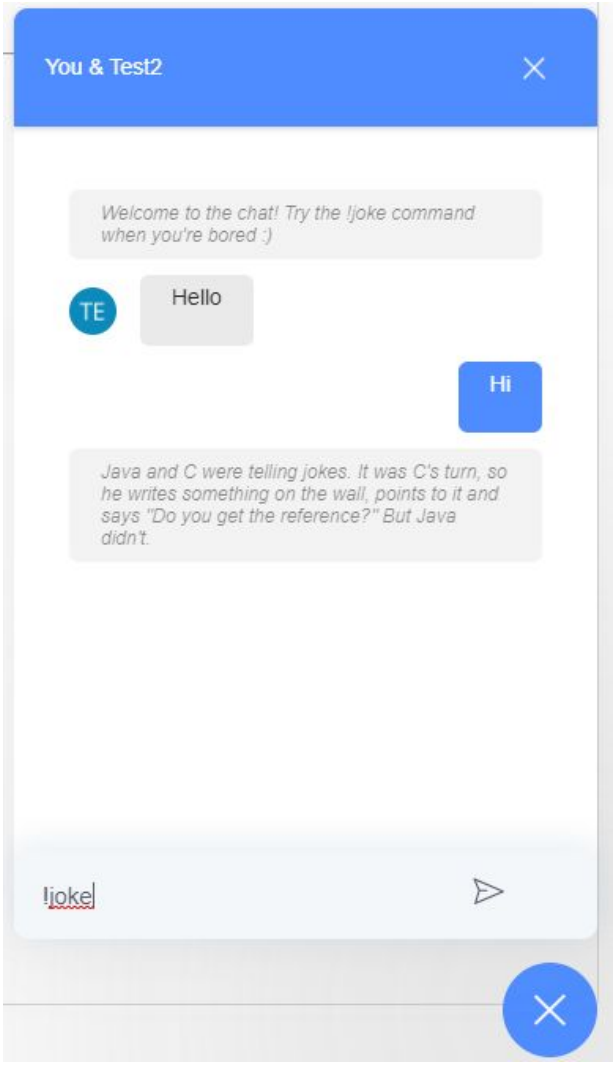
Der wesentliche Bestandteil der Callback-Funktion ist das Erstellen der entfernten bzw. eingefügten Strings in Zeile 148 und Zeile 157. Falls diese aus mindestens einem Char bestehen, werden sie der Liste hinzugefügt.
Zum Schluss wird die Debounce-Methode `sendContentDiffAfterTyping` aufgerufen.

```
34              sendContentDiffAfterTyping: debounce(e => {
35                  e.reduceContentChanges()
36                  e.sendWebSocketMessages()
37                  e.contentChanges = []
38              }, 400),
```

Eine Debounce-Methode besteht aus einer Callback-Funktion, die erst ausgeführt wird, sobald sie nach dem ersten Aufruf weitere X ms nicht aufgerufen wurde. Dadurch erhalten wir das Verhalten, dass die Methode erst ausgeführt wird, wenn für eine kurze Weile der Anwender keine Eingaben im Editor tätigt.
In der `sendContentDiffAfterTyping` Methode werden zunächst die Änderungen in der Liste komprimiert, um insbesondere das Schreiben eines Satzes in eine Änderung zu vereinen und dadurch nur einen WebSocket Event senden zu müssen.
Schließlich wird die Liste der komprimierten Änderungen nacheinander via WebSockets an den Server geschickt, wo der serverseitig State des Dokuments entsprechend transformiert bzw. synchronisiert wird.

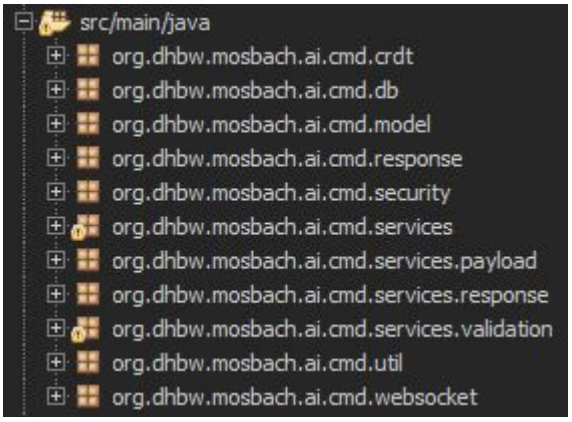
5.3 Chat

Mit dem eingebauten Chat können sich alle Beteiligten eines Dokuments unterhalten, ohne hierfür den Editor zu missbrauchen und auch ohne externe Anwendungen nutzen zu müssen.
Die Anbindung des Chats bedurfte kaum Anpassungen im Backend, da der bestehende WebSocket-Endpoint genutzt wird. Lediglich die neuen Eventtypen mussten ergänzt werden. Eine Persistierung des Chatverlaufs ist nicht vorgesehen.



6 Backend

Das Backend ist im MVC-Prinzip erstellt, wobei die DB-Zugriffe in die jeweiligen DAO-Klassen der JPA-Models ausgelagert wurden, gewisse Businesslogik in entsprechende Pakete verbracht (CRDT, Security) und die Controller als Steuerungspunkte diese mit dem Frontend verknüpfen (REST-API, Websocket Endpoint).



6.1 REST-API

REST (***Re**presentational **S**tate **T**ransfer*) ist ein Architekturentwurf für Web-Services, der von Roy T. Fielding im Jahre 2000 im Rahmen seiner Dissertation vorgeschlagen wurde und zustandslose Dienste beschreibt, welche zusammen eine einheitliche webbasierte Schnittstelle zu einer Anwendung formieren. Fielding schlägt den REST-Architekturstil als Leitmodell für die Funktionsweise des World Wide Web (WWW) vor, da seine Motivation in der Erschaffung eines neuen Modells für die Standards von Webprotokollen begründet liegt.

Das wichtigste Protokoll im Bezug auf REST ist das Hypertext Transfer Protocol (HTTP), das genau den Prinzipien von REST von Zustandslosigkeit, eindeutige Adressierbarkeit von Ressourcen und dem Client-Server-Modell entspricht. Dementsprechend kann ein REST-Service über die folgenden standardmäßigen HTTP-Methoden angesprochen werden:

- `GET`: eine oder mehrere Ressourcen werden vom Server an den Client geschickt.
- `POST`: legt eine neue Ressource am Server unter dem angegebenen Pfad an.
- `DELETE`: löscht eine Ressource auf dem Server unter dem angegebenen Pfad.
- `PUT`: aktualisiert die Ressource unter dem angegebenen Pfad.
- `PATCH`: aktualisiert einige Felder der Ressource unter dem angegebenen Pfad.

Mithilfe dieser Methoden und einer URL kann nun ein RESTful Web-Service aufgerufen werden, wobei der Web-Service dann weiß, welche Operation gewünscht wird. Obgleich die Interaktionen zwischen Client und Server als zustandslos gelten, kann der Server einen Cache anlegen, um wichtige Ressourcen wie beispielsweise eine Benutzersitzung (*User Session*) schneller laden zu können. Im Fall der hier implementierten API wird mithilfe des

Login-Dienstes eine User Session angelegt, um bei nachfolgenden Aufrufen sicherstellen zu können, dass ein Benutzer auch angemeldet ist.

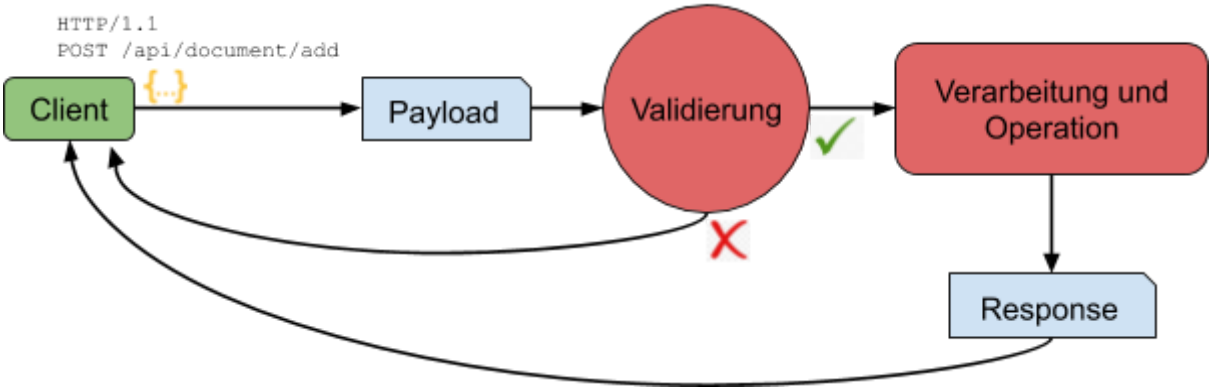
6.1.1 Komponenten der API

Damit das Frontend auf Dokumentendaten und Funktionalitäten zur Authentifizierung und Verwaltung der Dokumente zugreifen kann, wurde eine REST-API entwickelt, die sich in drei wesentliche Komponenten unterteilt:

- `AuthenticationService`: erlaubt die Authentifizierung eines Benutzers gegenüber der Anwendung. Dies umfasst den Login und die Registrierung neuer Benutzer sowie den Logout.
- `CollaboratorService`: ermöglicht die Hinzufügung und Entfernung von Kollaboratoren zu einem Dokument.
- `DocumentService`: bietet verschiedene Dienste zur Hinzufügung und Entfernung von Dokumenten, Rückgabe aller Dokumente des angemeldeten Benutzers sowie zur Übertragung der Besitztums eines Dokumentes an einen anderen Benutzer an. Daneben kann mithilfe dieses Dienstes abgefragt werden, ob der angemeldete Nutzer die Berechtigung hat, ein Dokument einzusehen und zu bearbeiten.

Zur Implementierung dieser REST-Endpoints wurde die vom WildFly mitgelieferte JAX-RS-Implementierung RESTEasy eingesetzt, die ebenfalls vom Unternehmen JBoss stammt. Die drei unterschiedlichen Endpoints sind von einem `RootService` abgeleitet, welcher den Basispfad zur REST-API festlegt und als Hauptinitialisierungspunkt der API fungiert. Der `RootService` selbst erweitert die JAX-RS `Application` Klasse und dient somit als Einstiegspunkt für alle weiteren Endpoints, die definiert werden. Darüber hinaus implementieren die REST-Endpoints das `RestEndpoint` Interface, welches als Verallgemeinerung der konkreten Implementierungen gedacht ist.

Die Komponenten, aus denen sich die API zusammensetzt, sind in der nachfolgenden Abbildung verdeutlicht.



Hierbei sendet ein Client als Benutzer der Anwendung eine Anfrage an den Server, wie z.B. die Erstellung eines neuen Dokumentes. Das Datenformat der Anfrage ist JSON. Beim Server wird die Anfrage durch einen JSON-Deserialisierer wie bspw. Jackson in ein Anfrageobkjekt umgewandelt, das hier als `Payload` bezeichnet wird. Die Anfrage wird durch eine Reihe von Validierungen überprüft und falls die Prüfung erfolgreich ist, wird die

Anfrage verarbeitet und die gewünschte Operation ausgeführt, z.B. das Dokument erstellt. Das Ergebnis wird dem Client als `ResponseObject` zurückgegeben, das bei der Versendung wieder in JSON umgewandelt wird. Falls die Validierung fehlschlägt, wird die Operation nicht ausgeführt und der Client erhält eine Antwort mit einer Fehlerbeschreibung.

Auf diese Weise macht das Frontend der Anwendung automatisch einige Aufrufe der API, um beispielsweise die Dokumente und Kollaboratoren für die Dokumentenliste in der Sidebar zu laden, einen Benutzer anzumelden oder zu registrieren.

6.1.2 Die Payload-Modelle

Der `Payload` ist ein leeres Interface, welches durch zahlreiche Klassen für die Abbildung von Anfragen implementiert wird. Konkret werden die Klassen zur Abbildung der JSON-Parameter auf die Objektattribute verwendet und sind daher notwendig, da die JAX-RS-Services kein direktes Mapping von JSON-Attributen auf Methodenparameter unterstützen. Die Attribute sind mit Jackson-Annotationen versehen, um die Zugehörigkeit zwischen JSON-Parametern und Attributen für den Server-Container kenntlich zu machen.

```
5   /**
6    * @author 6694964
7    * @version 1.2
8    */
9   public class RegisterModel implements Payload {
10
11      @JsonProperty(value = PayloadParameters.USERNAME, required = true)
12      private String username;
13
14      @JsonProperty(value = PayloadParameters.EMAIL, required = true)
15      private String email;
16
17      @JsonProperty(value = PayloadParameters.PASSWORD, required = true)
18      private String password;
19
20      public String getUsername() {
21          return username;
22      }
23
24      public String getEmail() {
25          return email;
26      }
27
28      public String getPassword() {
29          return password;
30      }
```

Es wurden hier bewusst keine JSON-B- oder XML-Annotationen benutzt, da die Verhaltensweise dieser Annotationen teilweise je nach Server-Konfiguration voneinander

abweichte. Im Code-Beispiel ist das Anfragemodell für die Registrierung zu sehen, die den Benutzernamen, die Email-Adresse und das Passwort für den neuen Benutzer entgegennimmt.

6.1.3 Die Validierung der Anfrage

In einem weiteren Paket `validation` befinden sich die Klassen für die Validierung der Anfragen. Jeder REST-Service ist hierbei mit einer eigenen Validierungsklasse assoziiert, die das Interface `ModelValidation<Payload>` implementiert und als Parameter das Payload-Objekt entgegennimmt. Die Validierung umfasst z.B. das Prüfen der Existenz von Dokumenten oder Benutzern, die über Ids oder Benutzernamen angegeben wurden, aber auch die Überprüfung der Rechte eines Benutzers, speziell die Prüfung, ob Benutzer Dokumente einsehen, bearbeiten oder löschen dürfen. Das untere Code-Beispiel ist dem Interface `ModelValidation` entnommen, welches die Basismethode `validate(T payload)` für alle Validierungsklassen definiert.

```
33      /**
34       * Checks the passed payload of the respective request model type for validity. The
35       * validation may include examinations for existence of resources such as documents
36       * and users, the presence of privileges to access a resource or perform a specific
37       * operation and the check for certain constraints required for this operation.
38       *
39       * The {@code validate} method follows the principle of error handling descending from
40       * the Go programming language. Accordingly, a potential error in the validation is
41       * returned as an unsuccessful {@link ValidationResult} using a sufficient error message
42       * rather than throwing an exception which requires additional mappers in the services
43       * to handle those exceptions. A successful validation is returned in the same manner
44       * as a successful validation result.
45       *
46       * In any case, the method should accept a non-null payload and return a non-null
47       * validation result. In case the payload should be {@code null} this method should
48       * answer with an unsuccessful validation result containing an internal server error
49       * response.
50       *
51       * @param payload the provided non-null payload to the service
52       * @return a non-null validation result indicating if the validation was successful
53       * or not
54       */
55      @NotNull
56      public ValidationResult validate(@NotNull T payload);
```

Die Validierungsmethoden arbeiten mit einem `ValidationResult` als Rückgabewert und folgen damit dem Beispiel der Fehlerbehandlung in der Programmiersprache Go. Anstatt eine Exception bei fehlgeschlagener Validierung zu werfen, wird stattdessen ein immutables `ValidationResult` Objekt erzeugt, welches das Ergebnis der Validierung mit einem Status und einer Fehlerbeschreibung enthält. Aus dem `ValidationResult` kann über eine Methode direkt die entsprechende Response für den Benutzer erzeugt werden, die dann vom REST-Service zurückgegeben wird.

```
120     @NotNull
121     public ValidationResult validate(@NotNull RegisterModel model) {
122         if (model == null) {
123             return ValidationResult.response(new InternalServerError("RegisterModel is null"));
124         }
125
126         final String username = model.getUsername();
127         final String email = model.getEmail();
128         final String password = model.getPassword();
129
130         final ValidationResult usernameCheck = checkUsernameConstraints(username);
131         if (usernameCheck.isInvalid()) {
132             return usernameCheck;
133         }
134
135         final ValidationResult emailCheck = validateEmailSyntax(email);
136         if (emailCheck.isInvalid()) {
137             return emailCheck;
138         }
139
140         final ValidationResult passwordCheck = checkPasswordConstraints(password);
141         if (passwordCheck.isInvalid()) {
142             return passwordCheck;
143         }
144
145         return ValidationResult.success("Registration was successful");
146     }
```

Das obenstehende Code-Beispiel ist die Implementierung der Registrierungsvalidierung. Es wird geprüft, ob das Format des Benutzernamens stimmt und dieser noch nicht registriert ist. Außerdem wird die Syntax der Email-Adresse verifiziert und geprüft, ob das Passwort einige Bedingungen erfüllt.

6.1.4 Die Verarbeitung der Anfrage

Nach einer erfolgreichen Validierung wird die Anfrage verarbeitet und durchgeführt. Das Payload-Objekt wird im eigentlichen Web-Service als Parameter übergeben wie auch ein Kontextobjekt für Daten aus der User-Anfrage. Im Beispiel der Registrierung wird nun ein neuer Benutzer angelegt, das Passwort verschlüsselt und der neue Benutzer in der Datenbank gespeichert. Schließlich wird eine erfolgreiche Antwort an den Client zurückgeschickt.

```
142     @POST
143     @Path("/register")
144     @Consumes(MediaType.APPLICATION_JSON)
145     @Produces(MediaType.APPLICATION_JSON)
146     @NotNull
147     public Response doRegister(@NotNull RegisterModel registerModel, @Context HttpServletRequest request) {
148         final ValidationResult registerCheck = registerValidation.validate(registerModel);
149         if (registerCheck.isInvalid()) {
150             return registerCheck.buildResponse();
151         }
152
153         final String username = registerModel.getUsername();
154         final String email = registerModel.getEmail();
155         final String password = registerModel.getPassword();
156
157         User newUser = new User();
158         newUser.setName(username);
159         newUser.setMail(email);
160
161         String hashedPassword = hashing.hashPassword(password);
162         newUser.setPassword(hashedPassword);
163
164         userDao.createUser(newUser);
165
166         log.info("{}: New user '{}' was registered successfully", request.getRequestURI(), username);
167         log.info("{}: Created new repository for user '{}'", request.getRequestURI(), username);
168         return new Success("Registration successful.").buildResponse();
169     }
```

6.1.5 Das Response-Modell

Sowohl bei fehlgeschlagener als auch bei erfolgreicher Validierung wird mit dem `ValidationResult` ein entsprechendes Response-Objekt angelegt, das vom REST-Service zurückgegeben wird. Das `ResponseObject` beinhaltet einen HTTP-Status mit der Zahl und der Beschreibung sowie eine Nachricht, die zusätzliche Informationen zur Verarbeitung der Anfrage liefert. Die REST-Endpoints können folgende HTTP-Status zurückgeben:

- `200 OK`: Die Anfrage wurde folgerichtig ausgewertet und die gewünschte Operation konnte ausgeführt werden.
- `400 Bad Request`: Die Anfrage beinhaltet ungültige Felder oder einige Bedingungen wurden nicht erfüllt.
- `401 Unauthorized`: Der Benutzer ist nicht berechtigt, die Operation durchzuführen. Zunächst muss er sich anmelden.
- `403 Forbidden`: Der Benutzer darf auf ein bestimmtes Dokument nicht zugreifen.

6.1.6 Die Benutzersitzung

Bei der Anmeldung eines Benutzers wird für ihn eine neue Sitzung (*Session*) erstellt. Diese enthält die Daten des Benutzers in Form eines User-Objektes und die Information, ob der Benutzer authentifiziert ist oder nicht. Die Verwaltung und Abfrage der Session wird über ein implementiertes `SessionUtil` vereinfacht, das Methoden zur Anlegung der Sitzung, Abfragen des Users und der Information, ob der User angemeldet ist oder nicht, anbietet. Außerdem kann es die aktuelle Sitzung des Benutzers beim Logout invalidieren.

Bei der Erstellung der Sitzung wird gleichzeitig ein Cookie `JSESSIONID` erzeugt, das die Sitzung referenziert. Dieses Cookie wird über den `Set-Cookie` HTTP-Header an den Client geliefert und muss bei jedem Aufruf eines REST-Services mitgegeben werden, um zu verifizieren, ob der anfragende Benutzer angemeldet ist.

6.2 Websockets

Um die Synchronisation der Dokumente, sowie des Chats zu erreichen, wird die Websocket-Technologie verwendet. Diese erlauben es uns, die Events in beiden Fällen asynchron zu erfassen und an die restlichen Teilnehmer zu publizieren.

Zur Abbildung des Vorgangs wurde ein eigenes Modell geschaffen, welches aus einer synchronisierten HashMap am Server besteht. Hier sind über die Dokumenten-Id als Key die aktiven Dokumente und deren aktuell aktive Nutzer erfasst.

```
/**
 * Active docs being worked on by n users
 */
private static Map<Integer, ActiveDocument> docs = Collections.synchronizedMap(new HashMap<>());
```

```
public class ActiveDocument {

    private Doc doc;
    private long state;
    private List<Session> users;
```

Hierzu wird beim Öffnen eines Dokuments über die Navigation im Frontend eine Verbindung aufgebaut. Nun wird geschaut, ob das Dokument generell existiert und möglicherweise bereits aktiv genutzt wird. In diesem Fall wird der neue Nutzer zur Liste der aktiven innerhalb dieses Dokuments hinzugefügt und er erhält den aktuellen Inhalt zurück.

Da es sich um eine SPA handelt, sind wir gezwungen die restlichen Informationen ebenfalls über asynchrone Wege an das Frontend zu senden. Um diesen Vorgang zu entkapseln, wurde das Message Objekt, so wie das Enum MessageType geschaffen.

```
public enum MessageType {

    Insert("Insert"),
    Delete("Delete"),
    UserJoined("UserJoined"),
    UserLeft("UserLeft"),
    ContentInit("ContentInit"),
    DocumentTitle("DocumentTitle"),
    UsersInit("UsersInit"),
    ChatMessage("ChatMessage"),
    WrongDocId("WrongDocId");
```

Damit lassen sich mit wenig Aufwand sämtliche Nachrichten, die benötigt werden, abbilden. Zudem ist das System extrem einfach erweiterbar, sollte ein weiterer Typ benötigt werden.

```java
@OnOpen
public void onOpen(@PathParam("docId") int docId, @PathParam(CmdConfig.SESSION_USERNAME) String username, @PathParam(CmdConfig.SESSION_USERID) int userId, Session session) {

    session.getUserProperties().put(CmdConfig.SESSION_USERNAME, username);
    session.getUserProperties().put(CmdConfig.SESSION_USERID, userId);

    Doc doc = null;

    if(docs.get(docId) == null) {
        doc = docDao.getDoc(docId);
        if(doc == null) {
            Message wrongDocIdMsg = messageBroker.createSystemMessage(userId, docId, String.valueOf(docId), MessageType.WrongDocId);
            messageBroker.publishToSingleUser(wrongDocIdMsg, session);
            return;
        }
        docs.put(docId, new ActiveDocument(doc, 0, new ArrayList<>()));
    }

    if(doc == null)
        doc = docs.get(docId).getDoc();

    if(doc.getContent() == null)
        docs.get(docId).getDoc().setContent("");

    docs.get(docId).getUsers().add(session);

    Message contentInitMsg = messageBroker.createSystemMessage(userId, docId, doc.getContent(), MessageType.ContentInit);
    messageBroker.publishToSingleUser(contentInitMsg, session);

    Message documentTitleMsg = messageBroker.createSystemMessage(userId, docId,  doc.getName(), MessageType.DocumentTitle);
    messageBroker.publishToSingleUser(documentTitleMsg, session);

    Message userInitMsg = messageBroker.createSystemMessage(userId, docId, messageBroker.getActiveUsers(docs.get(docId).getUsers(), session), MessageType.UsersInit);
    messageBroker.publishToSingleUser(userInitMsg, session);

    Message userJoinedMsg = messageBroker.createSystemMessage(userId, docId, messageBroker.formatUserMessage(session), MessageType.UserJoined);
    messageBroker.publishToOtherUsers(userJoinedMsg, docs.get(docId), session);
}
```

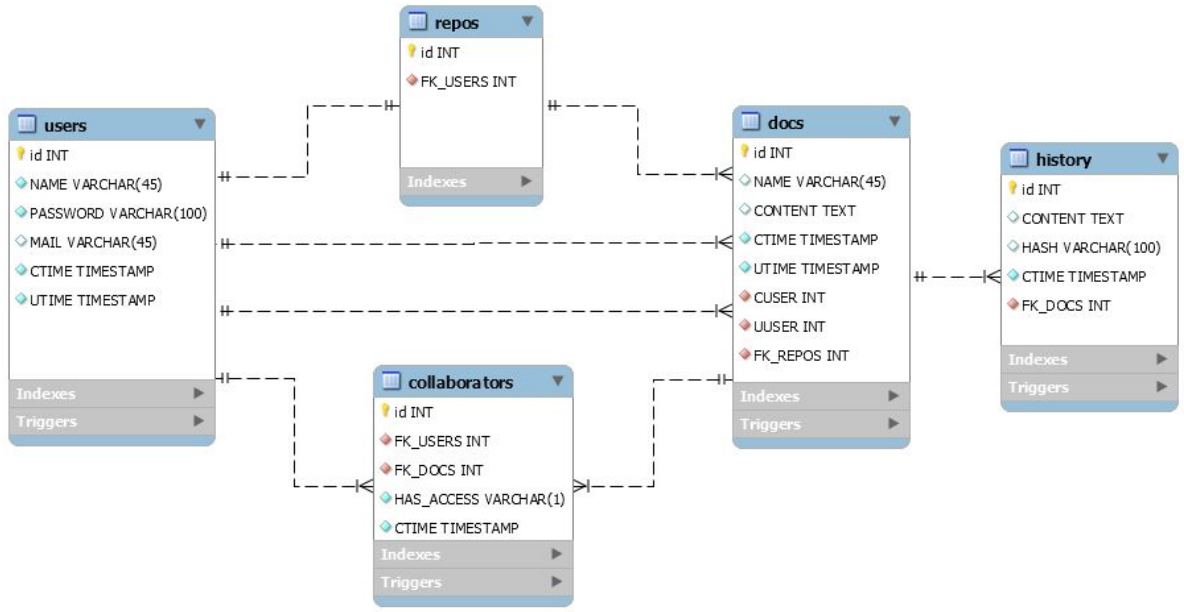
Über den MessageBroker wird eine Message dann, je nach Art und Zweck, ausschließlich an den User selbst, oder an die anderen Teilnehmer gesendet.

Diese MessageTypes erlauben es ebenso im onMessage-Fall jene Nachrichten, die eigentlichen Dokumenteninhalt widerspiegeln, von solchen, die für Organisatorisches zuständig sind, zu unterscheiden. Denn nur beim eigentlichen Dokumenteninhalt besteht der Bedarf diesen konsistent auf allen Clients zu halten, wodurch eine Transformation nötig wird. Im onClose-Fall erfolgen dieselben Schritte, wie im onOpen, allerdings in abgewandelter Reihenfolge. Der Nutzer wird aus der Liste der aktiven Nutzer eines bestimmten Dokuments herausgenommen. Diese Tatsache wird an die anderen Nutzer gemeldet. Sollten sich in einem Dokument keine aktiven Nutzer mehr befinden, wird der aktuelle Stand des Dokuments in die Datenbank persistiert, sowie der Eintrag aus der synchronisierten HashMap innerhalb des Websocket Endpoints entfernt.

```java
@OnClose
public void onClose(Session session) {

    for(int docId : docs.keySet()) {
        for(Session singleUserSession : docs.get(docId).getUsers()) {
            if(singleUserSession.equals(session)) {

                docs.get(docId).getUsers().remove(singleUserSession);

                int userId = (int) singleUserSession.getUserProperties().get(CmdConfig.SESSION_USERID);

                Message userLeftdMsg = messageBroker.createSystemMessage(userId,  docId, messageBroker.formatUserMessage(singleUserSession), MessageType.UserLeft);
                messageBroker.publishToOtherUsers(userLeftdMsg, docs.get(docId), session);

                break;
            }
        }

        if(docs.get(docId).getUsers().isEmpty()) {

            // Save current doc from db to history
            Doc currentDocState = docDao.getDoc(docId);
            History history = new History();
            history.setContent(currentDocState.getContent());
            history.setDoc(currentDocState);
            history.setHash(hashing.hashDocContent(history.getContent()));
            historyDao.createHistory(history);

            // Save current doc from client to db
            Doc activeDoc = docs.get(docId).getDoc();
            activeDoc.setUuser(userDao.getUserByName(session.getUserProperties().get(CmdConfig.SESSION_USERNAME).toString()));
            docDao.updateDoc(activeDoc);

            docs.remove(docId);
        }
    }
}
```

6.3 Datenbankanbindung und -struktur

Die Datenbankanbindung wurde via JPA realisiert. Ein Modell der Datenbank, erstellt im MySQL Workbench, liegt ebenfalls bei. Zur einfacheren Nutzung wurden Trigger direkt in die Modelle geschrieben, um das separate Aufsetzen der Datenbank via Skript nicht nötig zu machen.



Die Datenbankstruktur ist relativ simpel. Es wurde ein Schema entwickelt und anschließend normalisiert. Die folglich erarbeiteten Tabellen wurden anschließend als Modelle innerhalb des Programmcodes unter Berücksichtigung der Beziehungen erstellt.

6.4 Editor-Synchronisation

Außer der schnellen Übertragung der Nachrichten zwischen den vielen (möglichen) Clients muss natürlich auch gewährleistet werden, dass der Inhalt des Editors auch konsistent am Server sowie den Clients selbst vorhanden ist.

Um die Latenz zu reduzieren, werden Nachrichten zunächst direkt an den Client zurück geschickt. Da von diesem aus das ursprüngliche Event geschickt wurde, weiß er, was zu tun ist und wie die Nachricht in das Dokument eingebracht werden kann.

Im Backend wird nun die eingegangene Nachricht transformiert, sowie in den konsistenten Stand des Dokuments der globalen Variable im Websocket-Endpoint gespeichert.

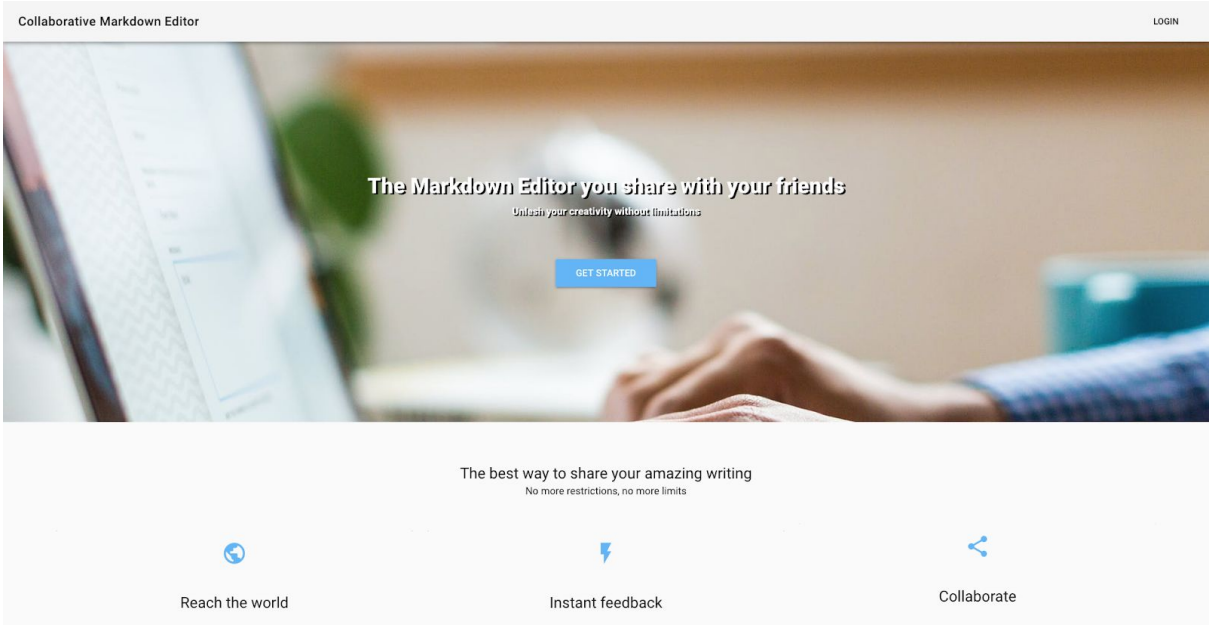
Darüber hinaus wird die ausgeführte Operation bzw. die zugehörige Nachricht in einer Nachrichtenliste gespeichert. Diese Liste besteht aus den letzten 10 Vorgängen und wird bei inkonsistenten Ständen von Clients dazu genutzt, beim Client den konsistenten Zustand herbeizuführen.

Dazu wird über die Liste, in der korrekten Reihenfolge, ab dem Zeitpunkt der Inkonsistenz iteriert und jeweils die dort getätigte Operation auf die Nachricht angewandt.

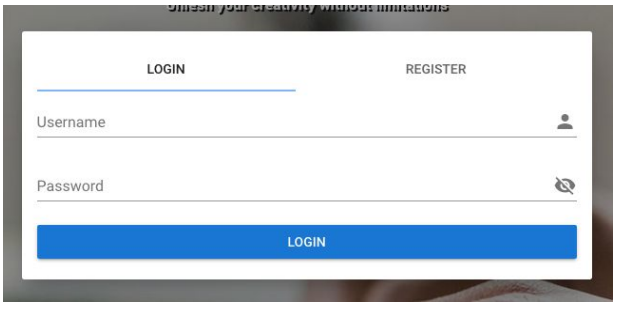
7 Benutzung der Weboberfläche

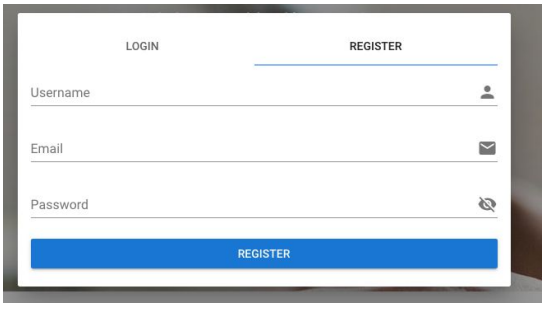
7.1 Anmeldung und Registrierung

In der folgenden Abbildung ist die Darstellung der Website beim erstmaligen Aufrufen gezeigt. Über die “Login” Schaltfläche öffnet sich der Dialog zum Anmelden und Registrieren.



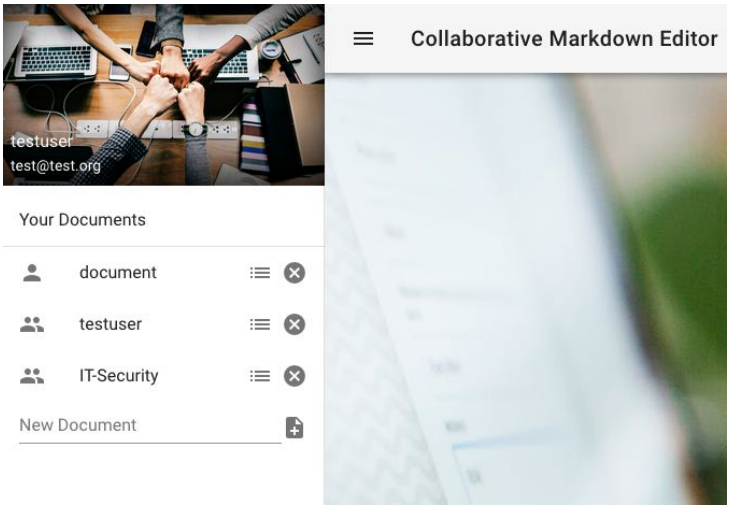
Über die nun folgenden Eingabefelder kann der Benutzer sich initial registrieren und danach mit seinem Benutzerprofil anmelden. Hierbei findet eine Validierung der Eingaben statt, damit valide Benutzernamen, Email-Adressen und sichere Passwörter erstellt werden.

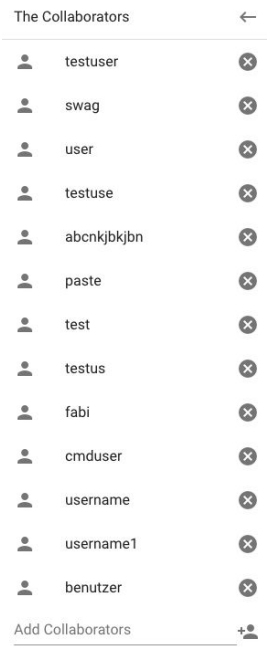




Wenn man angemeldet ist, öffnet sich eine Sidebar auf der linken Seite des Browser-Fensters, in der alle Dokumente des Benutzers angezeigt werden. Wenn man auf diese klickt, kommt man in das Editor-Fenster und kann das jeweilige Dokument bearbeiten. Das Icon vor dem Dokument deutet an, ob eine oder mehrere Personen an dem Dokument arbeiten können, d.h. ob Kollaboratoren hinzugefügt wurden oder nicht.

7.2 Die Sidebar

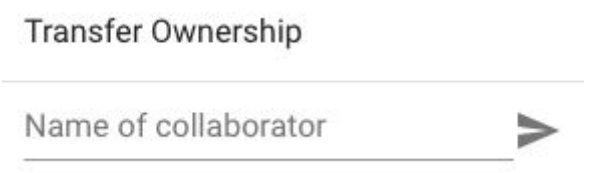




Über das Listensymbol kommt man auf die Liste der Kollaboratoren und als Eigentümer des Dokumentes kann man über das Eingabefeld unten neue Kollaboratoren hinzufügen. Dazu muss der Benutzername des Kollaborators eingetragen werden. Als Nicht-Eigentümer ist man nicht berechtigt, Kollaboratoren hinzuzufügen oder zu entfernen. Versucht man dies, erscheint eine Fehlermeldung als Notification.

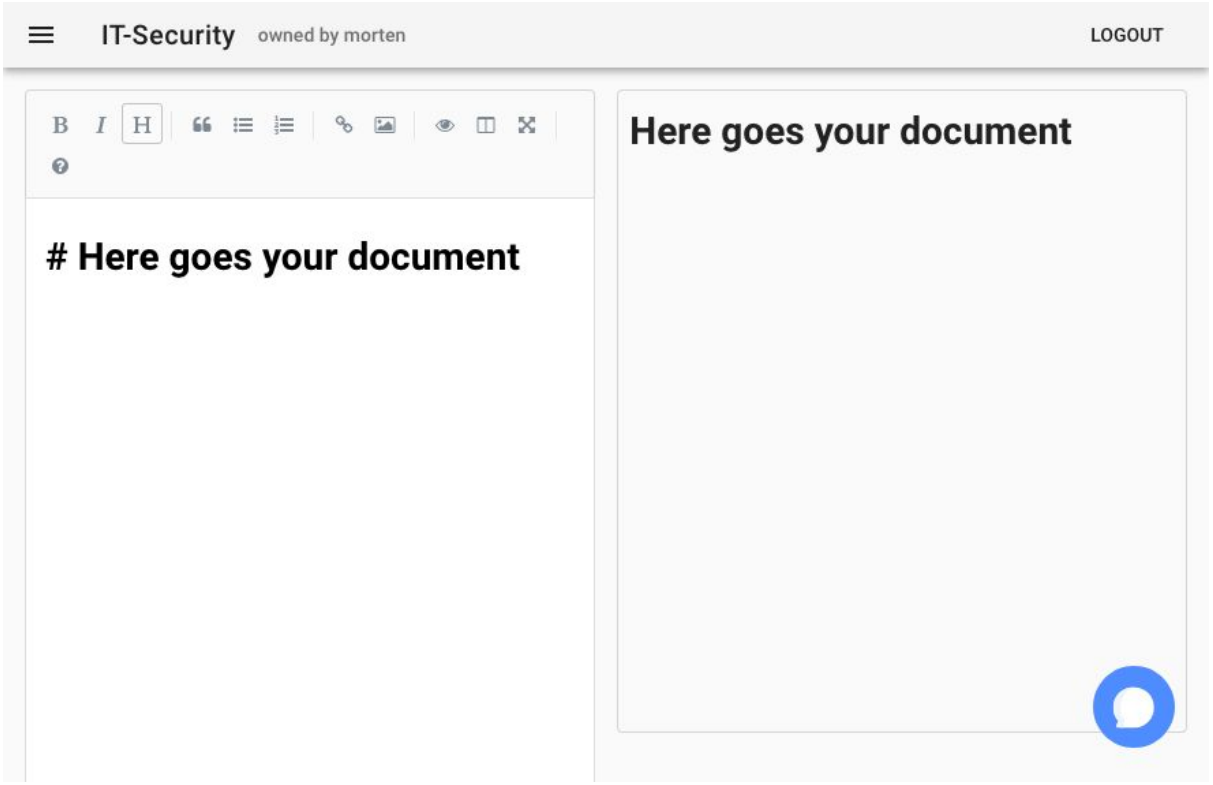
Jedoch kann man sich selbst aus dem Dokument austragen, indem man das Kreuz neben seinem Kollaboratornamen anklickt. Mit diesem Klick verliert man aber auch den Zugriff auf das Dokument.

Außerdem kann man in dieser Ansicht als Eigentümer des Dokumentes das Besitztum am Dokument auf einen anderen Benutzer übertragen. Auch hier muss dann der Nutzername des neuen Eigentümers eingegeben werden. Jedoch kann man nur einen aktiven Kollaborator des Dokuments zum neuen Eigentümer benennen.

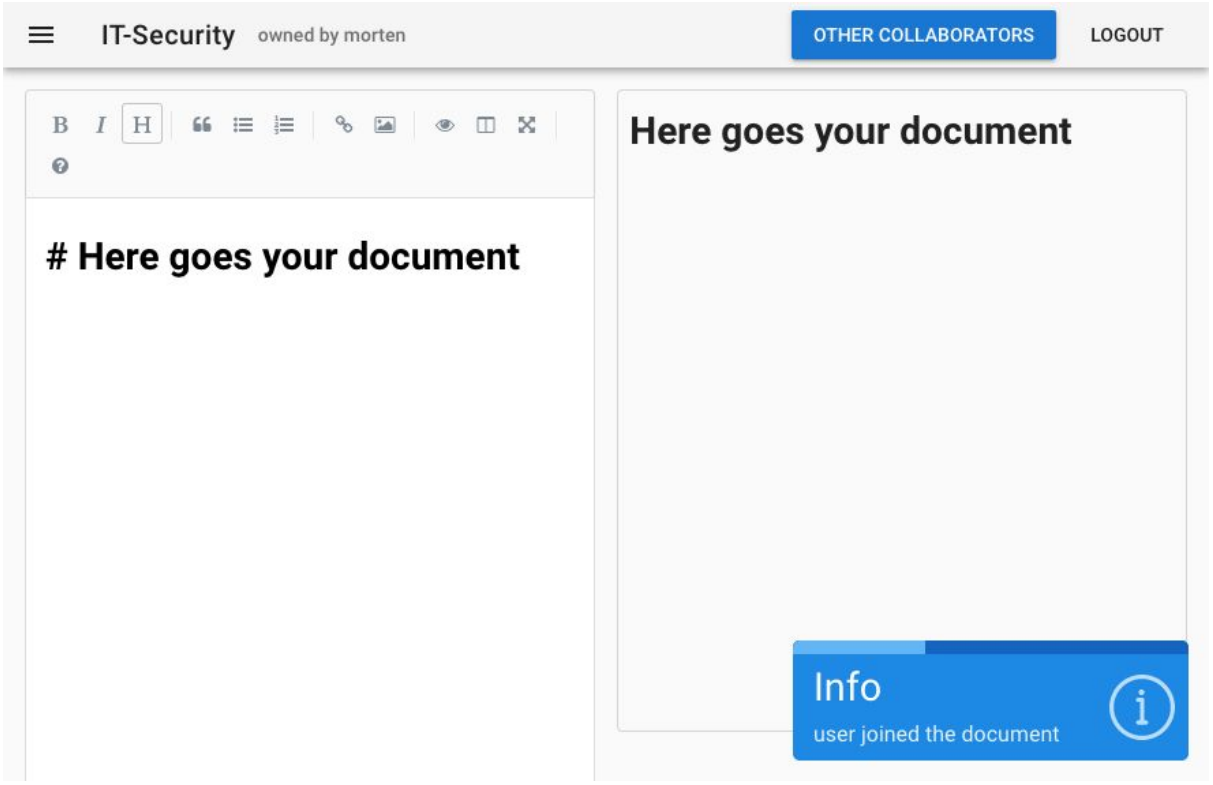


7.3 Das Editorfenster

Der Editor teilt sich in die Bereiche der Editorfläche auf der linken Seite und der live gerenderten Vorschau auf der rechten Seite. In der Editorfläche kann das Dokument bearbeitet werden, das mithilfe der Markdown-Syntax formatiert wird. Wichtige Formatierungen können in der Editor-Toolbar eingefügt werden. Je nach Dokument ändert sich die Vorschau auf der rechten Seite. Unten rechts in der Ecke befindet sich der Chat, mit dem man mit anderen Kollaboratoren, die gerade in dem Dokument angemeldet sind, kommunizieren kann.



Sobald ein anderer Benutzer dem Dokument beitritt, bekommen alle anderen Benutzer eine entsprechende Benachrichtigung. Dasselbe gilt, falls ein Benutzer das Dokument verlässt. Die Namen der anderen Kollaboratoren, die sich im Dokument befinden, kann man über die Schaltfläche “Other Collaborators” sehen. Außerdem sind sie im Chat eingeblendet.



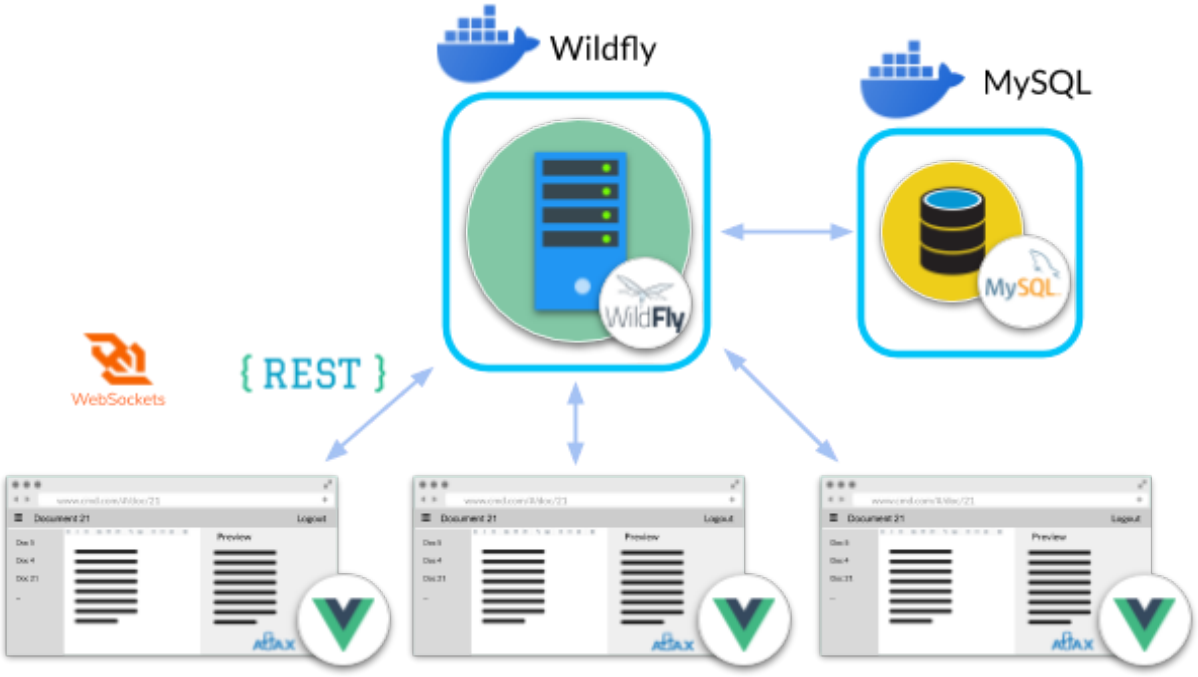
7.4 Die Chat-Komponente

Durch einen Klick auf die Chat-Schaltfläche öffnet sich das Chat-Fenster, in dem man Nachrichten an andere Kollaboratoren im Dokument schreiben kann, damit nicht das Dokument dazu zweckentfremdet werden muss. Oben im Titel stehen außerdem nochmal alle Namen der Kollaboratoren aufgelistet. Neben normalen Chat-Nachrichten unterstützt der Chat auch Chat-Befehle. Der momentan einzige Chat-Befehl ist `!joke`, womit ein Programmierwitz ausgegeben wird.

Über die “Logout” Schaltfläche kann man sich wieder vom System abmelden und man wird zurück auf die Startseite weitergeleitet. Versucht man, ein Dokument ohne Zugangsberechtigung zu öffnen, erscheint eine 403 Forbidden Page. Ein nicht existenter Pfad wird mit einer 404 Not Found Seite beantwortet.

8 Deployment

Die einfachste Methode, die Anwendung zu deployen, ist sicherlich die bereits vorhandenen Docker-Images zu nutzen, da sie die komplette Konfiguration einschließlich der Kompilierung der Anwendung und dem Deploy auf den Webserver automatisieren. Dabei sind die Container wie in der folgenden Abbildung angedeutet beim Wildfly und bei MySQL angeordnet. Die Kommunikation findet auf Basis eines eigenen Docker-Netzwerkes statt. Sowohl Wildfly als auch MySQL können auf getrennten Servern laufen und dynamisch skaliert werden. In dieser Hinsicht ist Docker sehr flexibel.



8.1 Portainer

So ist es uns gelungen, unsere Anwendung erfolgreich auf einen Portainer-Server zu deployen. Diese findet man unter dem Link http://10.50.12.73:32798/CMD. Dafür muss man allerdings mit dem Netzwerk der DHBW Mosbach verbunden sein, was auch per VPN möglich ist.

8.2 Heroku

Ein weiterer Dienst ist Heroku, welches eine Cloud Application Platform als Platform as a Service (PaaS) anbietet. Hier war es nicht so einfach, die Anwendung mit Docker zu deployen. Zwar ist Heroku durchaus in der Lage, über eine `heroku.yml` Datei Docker-Images zu bauen, aber kann es kein Docker-Netzwerk erstellen, weshalb der Wildfly-Server die MySQL-Datenbank nicht finden konnte (d.h. `UnknownHostException`).

Auf Heroku ist es so, dass jede Komponente auf einem eigenen Server läuft, die *Dynos* genannt werden. Jeder Dyno definiert hierbei einen Befehl, der ausgeführt wird, um die Anwendung oder Datenbank zu starten. Daneben bietet Heroku eine Reihe an Addons für Dynos, die hinzugefügt werden können, um bestimmte Dienste anzuknüpfen. Leider ist darunter keine kostenfreie MySQL-Lösung, weshalb wir hier auf einen kostenlosen Plan einer PostgreSQL-Datenbank wechseln mussten.

Heroku ist generell zum Aufbau von vernetzten Systemen, Cloud-Anwendungen und Orchestrierung von Containern geeignet. Die Server werden basierend auf der Infrastruktur von Amazon AWS ausgeführt und bekommen eine URL zugewiesen, über die der Netzwerk-Traffic läuft.

Für unsere Anwendung haben wir uns schließlich für die Implementierung eigener Buildpacks entschieden, die eine ähnliche Konfiguration wie bei den Docker-Containern vornehmen, nur nicht mit Docker funktionieren. Die Buildpacks werden zur Anwendung hinzugefügt und bauen zunächst die Anwendung mit Backend und Frontend durch Maven, laden sich dann einen Wildfly herunter und konfigurieren diesen mit PostgreSQL-Konnektor und entsprechender Datasource. Die Buildpacks befinden sich hier:

- https://github.com/mortenterhart/heroku-buildpack-wildfly
- https://github.com/mortenterhart/heroku-buildpack-wildfly-postgresql

Der finale Deploy der Anwendung befindet sich unter
https://collaborative-markdown-editor.herokuapp.com.

Hinweis: Der Dyno, auf dem der Wildfly-Server läuft, wird aus Kostengründen nur dann hochgefahren und gestartet, wenn ein Benutzer versucht, die Website zu erreichen. Wenn das versucht wird, wird der Server aus dem Standby-Modus aufgewacht und gestartet. Daher kann es vorkommen, dass man ein bisschen warten muss, bis eine Website geladen wird, selbst wenn es nicht den Anschein macht, als ob etwas lädt.

9 Zukunftsaussichten

Unser kollaborativer Markdown-Editor bietet bereits alle Basisfunktionalitäten, um damit produktiv in einem Team arbeiten zu können. Dennoch sind uns während der Entwicklung weitere Nice-to-Have-Features eingefallen, die zukünftig eingebaut werden können:

- Versionierung von Dokumenten im Editor
- Iterative Speicherung von Dokumentenständen
- Möglichkeit den Zugriff zu einem Dokument über ein Link zu verteilen
- Zusätzliche Authentifikation via Google, Facebook, etc. ermöglichen
- Dokumente mit einem Permalink veröffentlichen
- Unterstützung mathematischer Formeln
- Bessere Dokumentenübersicht z. B. mithilfe von Tags
- Export von Dokumenten nach PDF und HTML
- Verwaltung von hochgeladenen Bildern auf unserem Webserver

Ein paar dieser Features existieren schon in Ansätzen wie beispielsweise die bisher nicht verwendete History-Tabelle in der Datenbank. Diese ist für das Abspeichern bestimmter Dokumentenstände vorgesehen, die in regelmäßigen Zeitintervallen durch den WebSocket-Endpoint ausgelöst werden. So wäre es auch möglich, auf eine frühere Version eines Dokumentes zurückzuspringen.