

Dokumentation

T4-Documents

Lukas Köhler 7960359
ON22A-3, November 2023

Link zum GitHub Repository: [T4-Document-Application-GitHub](#)

Inhaltsverzeichnis

1 Thema des Projektes	2
2 Ausgangssituation.....	2
2.1 Skill Übersicht	2
3 Vorgehen.....	3
4 Anforderungsliste	4
5 Konzeption	4
5.1 Technologie / Werkzeugauswahl	4
5.2 Entwurf	5
5.2.1 Database Singleton	5
5.2.2 AuthenticationProvider	6
5.2.3 Der WebSocket Server	6
6 Ergebnis des Projekts	7
7 Reflexion	11
7.1 Herausforderungen	11
7.2 Unterstützung	12
7.3 Lernerfolge / Fazit.....	12
A Installationsanleitung	12
B Benutzerhandbuch	13
Use Case 1: Ein Dokument erstellen und teilen	13
Use Case 2: Ich möchte meine Daten ändern	14
Use Case 3: Ich möchte meinen Account löschen	14

1 Thema des Projektes


In diesem Projekt soll eine Webanwendung realisiert werden, welche es erlaubt, Dokumente mittels eines CRUD Schemas zu verwalten, diese zu teilen und in Echtzeit mit mehreren Nutzern zu editieren. Ich finde ein solches Projekt besonders interessant, da Echtzeitkommunikation zeitempfindliches Handling von Daten erfordert und spannend zu implementieren ist.

Konkrete Features der Anwendung sollen sein:

- Dokumente erstellen
- Dokumente umbenennen
- Dokumente löschen
- Dokumente lesen
- Dokumente editieren
- Benutzer registrieren
- Benutzerdaten ändern
- Benutzeraccount löschen
- Passwort ändern
- Dokumente teilen
- Dokumente in Echtzeit editieren (mit mehreren Nutzern)

Das Hauptaugenmerk der Anwendung besteht in der Echtzeitkommunikation, welche mithilfe eines WebSocket Servers realisiert wird.

2 Ausgangssituation

Ich habe bereits viel Erfahrung mit der Entwicklung von Web- sowie Desktopanwendungen gemacht. In der Praxis bei UEBERBIT verwenden wir für die Erstellung von Anwendungen zumeist PHP mit TYPO3 als CMS. Allerdings programmiere ich bereits, seit ich 14 Jahre alt bin. Angefangen habe ich mit Desktop Anwendungen in C/C++ und Qt. Im Laufe der Zeit habe ich mich mit vielen weiteren Programmiersprachen beschäftigt, darunter Python, JavaScript und C# mit .NET. In der Webentwicklung mit ich vergleichsweise neu, mit Vue.js arbeitete ich hier das erste Mal.  Mein grundsätzlicher Skill-Schwerpunkt ist derzeit noch im Backend Development sowie DevOps. Vor kurzem eignete ich mir noch Wissen über Docker an, um Deployments konsistenter und einfacher zu machen. In RESTful APIs bin ich zudem moderat erfahren (es ist nicht die Erste).

Daraus ergibt sich, dass ich vor allem in objektorientierter Programmierung bereits erfahren bin und ein Sprachen-unabhängiges Verständnis für den „Fluss der Daten“ habe. Dennoch gebe ich zu, dass mein Code bislang, was Struktur, Testing und Design Patterns angeht, noch an „Panzertape-Code“ erinnert. Mein generelles Ziel ist es, meinen Code sauberer und besser zu strukturieren. Das ist etwas, woran ich noch arbeite. Auch meine Kompetenzen mit JS-Frameworks möchte ich mit der Zeit ausbauen.

Eines meiner Prinzipien ist, dass man nicht immer den neuen „hot stuff“ verwenden muss, aber dennoch nicht von Entwicklungen abgehängt werden darf.

2.1 Skill Übersicht

1. PHP
2. MySQL
3. Docker & Docker Compose
4. JSON / YAML

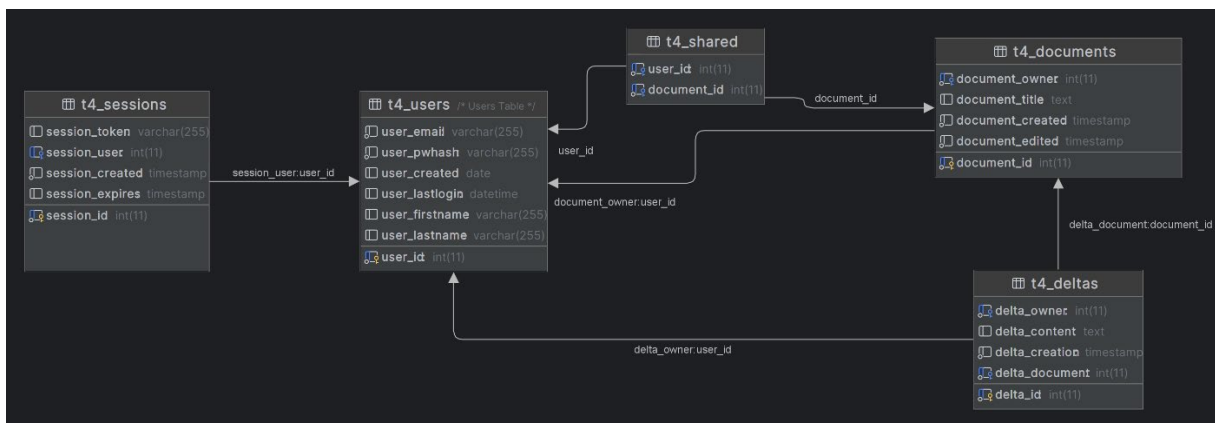
5. C/C++
6. C#
7. REST-APIs
8. TYPO3
9. Deployments auf Linux Server
10. HTML/CSS/JS

3 Vorgehen

Das Vorgehen ist recht unkompliziert gestaltet. Man könnte es sich als eine Checkliste vorstellen: Zunächst habe ich mir überlegt, wie überhaupt ein Rich Text Editor in Vue.js zu implementieren ist. Nach einiger Recherche im Internet kam ich zu VueQuill, welcher ein recht beliebter Editor für Vue.js ist.

Bevor ich die Datenbank entwerfen konnte, musste ich prüfen, wie dieser Editor Dokumente handhabt. Ich fand durch die Dokumentation heraus, dass sogenannte Deltas verwendet werden. Ein Delta ist beispielsweise ein eingegebenes Zeichen. Eine Menge Deltas ergeben dann das vollständige Dokument. Das bedeutet für die Datenbank, dass es eine beliebige Menge Deltas für ein Dokument geben kann, welche dieses ausmachen. Die Deltas müssen dem Dokument zugeordnet sein.

Mit diesem Wissen konnte ich ein Datenbanklayout erstellen:



In diesem Layout wurden alle Aspekte der Applikation bedacht, insbesondere das Teilen von Dokumenten mit einer M:N Beziehungstabelle sowie Session-Management via Tokens.

Die Implementation erfolgte in folgenden Schritten:

1. Implementation der REST-API
2. Implementation des Realtime-Servers
3. Implementation des Vue.js Frontends

Die API wurde mithilfe von HTTPie bereits vor der Erstellung des Frontends auf ihre Funktion hin geprüft, sodass man von der Modularität dieses Aufbaus profitieren kann (Frontend unabhängig vom Backend).

Für die API habe ich zusätzlich eine Swagger Open API 3.1 Spezifikation geschrieben, welche alle Parameter sowie deren Antworten dokumentiert. An dieser konnte ich mich im Entwicklungsprozess orientieren.

4 Anforderungsliste

Im Folgenden wird aufgelistet, welche Features geplant waren und welche davon umgesetzt wurden:

Nr.	Feature	Bereich	Status
1	Dokumente erstellen	REST-API	Umgesetzt
2	Dokumente umbenennen	REST-API	Umgesetzt
3	Dokumente löschen	REST-API	Umgesetzt
4	Dokumente teilen	REST-API	Umgesetzt
5	Dokumente lesen	REST-API	Umgesetzt
6	Echtzeit-Editor	Realtime-Server	Umgesetzt
7	Registrieren	REST-API	Umgesetzt
8	Einloggen	REST-API	Umgesetzt
9	Account löschen	REST-API	Umgesetzt
10	Accountdaten bearbeiten	REST-API	Umgesetzt
11	Passwort ändern	REST-API	Umgesetzt
12	Delta-CRUD	REST-API	Umgesetzt, nur notwendig, wenn WebSocket Verbindung nicht möglich.
13	Short-Polling via REST	Vue-Frontend	Nicht umgesetzt, da nicht notwendig. Endpunkte bleiben erhalten.
14	WebSocket Sync für Dokumente	Vue-Frontend	Umgesetzt
15	Login-Form	Vue-Frontend	Umgesetzt
16	Register-Form	Vue-Frontend	Umgesetzt
17	Komponente RTE	Vue-Frontend	Umgesetzt
18	Komponente ProfileView	Vue-Frontend	Umgesetzt
19	Komponente DocumentSidebar	Vue-Frontend	Umgesetzt
20	App.vue	Vue-Frontend	Umgesetzt

5 Konzeption

5.1 Technologie / Werkzeugauswahl

Für die Erstellung der Applikation habe ich, wie durch den Projektrahmen bereits fest vorgegeben, Vue.js sowie PHP verwendet. Im Falle der REST-API versuchte ich, ohne Composer-basierte Libraries klarzukommen, um maximale Kontrolle über den Code zu haben. Für den Realtime-Server war zwingend eine Library für das WebSocket Protokoll notwendig, weshalb selbiger Composer als Package Manager verwendet wurde.

Für den gesamten Entwicklungsprozess habe ich Docker sowie die Entwicklungstools von JetBrains verwendet. Einer der vielen Gründe für die Nutzung der JetBrains Software ist für mich die extrem gute Code-Completion, welche ebenfalls Datenbanklayouts berücksichtigen kann. In den JetBrains IDEs verfügte ich zudem über alle Tools, die ich benötigte, ohne etliche Plugins installieren zu müssen. Mit dieser Funktion konnte ich schneller arbeiten als bspw. in VSCode. Konkret verwendete ich die IntelliJ IDEA Ultimate, mit welcher man annähernd alle Sprachen programmieren kann. Für das Datenbank Layout verwendete ich die Datenbank-Lösung DataGrip, ebenfalls von JetBrains. Mit dieser konnte ich in hohem Tempo mein Datenbank-Layout entwerfen und auf den Docker-Server übertragen, Testdaten anlegen usw.

Lukas Köhler

ON22A-T4

Entwicklung einer Webapplikation Vue.js/PHP

Für das Testing der API verwendete ich den API-Testing-Client HTTPie, welcher es mir erlaubte, alle Endpunkte der API vorab zu testen. Damit konnte ich API / Server / Frontend beinahe vollständig getrennt voneinander entwickeln.

5.2 Entwurf

Das Projekt T4-Documents besteht im Grunde genommen aus 3 großen Komponenten, welche für die Funktion unerlässlich sind:

1. Eine REST-API, welche Interaktion mit der Datenbank erlaubt
2. Ein WebSocket-Server, welcher Live-Sessions zwischen den Usern behandelt
3. Ein Vue.js Frontend als SPA, welcher beide Komponenten nutzt.

Zunächst könnte man die Überlegung anstellen, dass es doch eigentlich reicht, nur einen WebSocket Server zu bauen und diesen als Handler für alles zu verwenden.

Theoretisch, ja, praktisch würde das den WebSocket Server maßgeblich komplizierter machen und die durchgängige Verbindung mit dem Server mit Dingen belasten, die nicht zeitempfindlich sind. Das kann die Performance der Applikation stark beeinträchtigen. Daher entschied ich mich dafür, lediglich Live-Synchronisation mit dem WebSocket zu bauen und alles andere via REST.

Die REST-API wird für alle üblichen Operationen verwendet: Login / Session-Management, CRUD mit Dokumenten.

5.2.1 Database Singleton

Link zum Code: [t4-document-application/api/src/DatabaseSingleton.php at main · lukkoeh/t4-document-application \(github.com\)](https://github.com/lukkoeh/t4-document-application/blob/main/src/DatabaseSingleton.php)

Die erste Klasse, welche ich sowohl für meine REST-API als auch für meinen WebSocket Server erstellt habe, ist ein Datenbank-Singleton. Hier habe ich, wie es der Name bereits sagt, das Singleton Designpattern verwendet. Sobald die Klasse instanziiert wird, wird entweder die Instanz erstellt oder die aktuelle Instanz der Datenbankverbindung zurückgegeben. Speziell in PHP musste ich hier folgende Punkte beachten:

- Sofern nicht explizit beachtet, kann eine Instanz einer Klasse geklont werden mittels `__clone()`. Sofern die möglich bleibt, ist die Klasse per Definition kein Singleton mehr. Daher habe ich `__clone()` so modifiziert, dass eine Exception erstellt wird, sofern man versucht den Singleton zu klonen.
- Der Konstruktor `__construct()` ist `private`, eine neue Instanz der Klasse kann nicht über den gewohnten Weg via „new“ erstellt werden.

Um also den Singleton zu vervollständigen, habe ich noch eine statische Methode mit Namen `getInstance()` implementiert, welche wie folgt aufgerufen werden kann (einziger Weg, eine Datenbank-Instanz zu bekommen):

```
DatabaseSingleton::getInstance();
```

Zur Interaktion mit der Datenbank habe ich zwei simple Funktionen erstellt, welche wie ein Interface bindend sind:

- `perform_query($query, $params)` – bekommt eine rohe Query und Parameter als Array und führt alles als prepared statement aus.
- `get_last_inserted_id()` – Gibt die zuletzt eingesetzte ID eines Datensatzes zurück. Gut, um bspw. `session_id` nach dem Einsetzen eines neuen Tokens zu finden.

Link zum Code: [t4-document-application/api/src/AuthenticationProvider.php at main · lukkoeh/t4-document-application \(github.com\)](https://github.com/lukkoeh/t4-document-application/blob/main/api/src/AuthenticationProvider.php)

Der AuthenticationProvider ist eine Klasse, welche alle Funktionalitäten bietet, welche für User / Session Management erforderlich sind. Im Rahmen dieser Dokumentation gehe ich nur auf die wichtigsten Funktionen ein:

Mithilfe der Login-Funktion werden Username und Passwort validiert und ein Token für den Account erstellt. Standardmäßig wird ein Token mit einer Gültigkeit von einem Tag erstellt. Dieser Token kann dann für andere Operationen verwendet werden.

Es gibt zudem noch die statische Methode `validateToken($token, $needs_response = false)`. Diese validiert einfach einen Token. Dabei wird berücksichtigt, ob es den Token gibt und ob er abgelaufen ist. Die restlichen Funktionen bilden einen klassischen CRUD Zyklus für User ab und werden hier nicht genauer behandelt. Weitere Informationen zu den API Endpunkten für User können Sie der Swagger Open API 3.1 Spezifikation entnehmen, welche wahlweise als YAML in API-Definition.yaml im API Ordner vorhanden ist, oder bei laufender App unter `localhost:10004` erreichbar ist.

5.2.3 Der WebSocket Server

Link zum Code: [t4-document-application/server/src/WebSocketServer.php at main · lukkoeh/t4-document-application \(github.com\)](https://github.com/lukkoeh/t4-document-application/blob/main/server/src/WebSocketServer.php)

Sobald die App geöffnet wird, wird parallel eine WebSocket Verbindung zum Server aufgebaut. Wenn ein Dokument ausgewählt wird, so wird dem Server gemeldet, welches Dokument editiert wird. Sofern der User authentifiziert ist (Token wird mitgesendet), wird die Verbindung mit dem Client in ein mehrdimensionales Array einsortiert.

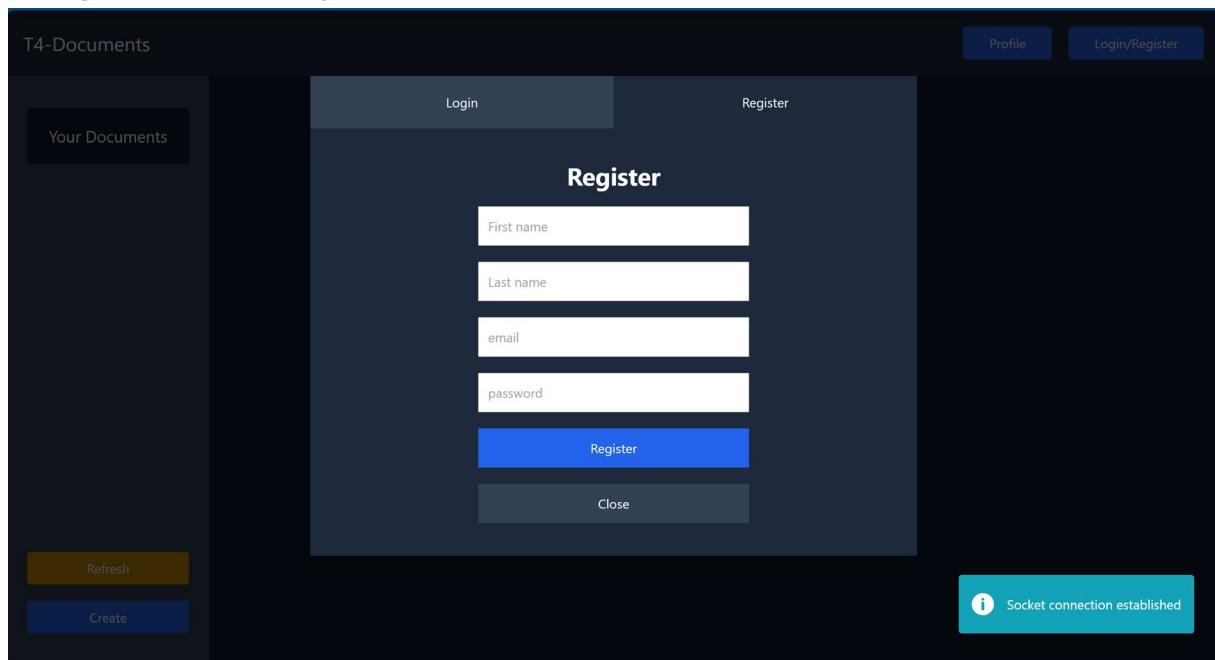
```
<?php
```

```
$connections = [  
    1 => [$connection1, $connection2, $connection3],  
    2 => [$connection1, $connection2, $connection3],  
    3 => [$connection1, $connection2, $connection3],  
];
```

Das obere Beispiel zeigt das System in einer beispielhaften Weise. Die Keys im assoziativen Array spiegeln die Dokument-ID wider, welche aktuell editiert wird. Diesem Dokument werden dann die Verbindungen der Clients zugeordnet. Sendet nun ein Client ein Delta für ein Dokument, so wird dieses nicht nur, wie in einer REST-API, in die Datenbank gespeichert. Es wird zusätzlich an alle Verbindungen gesendet, welche zuvor vermeldet haben, dass sie dieses Dokument editieren.

Wird eine Verbindung getrennt, wird der Client aus allen Connection Pools entfernt. Meldet er ein neues Dokument, wird er in einen neuen Connection-Pool einsortiert und aus allen anderen entfernt.

Dieses Layout erlaubt es dem WebSocket Server mit einer einzigen Instanz alle Live-Sessions behandeln zu können.

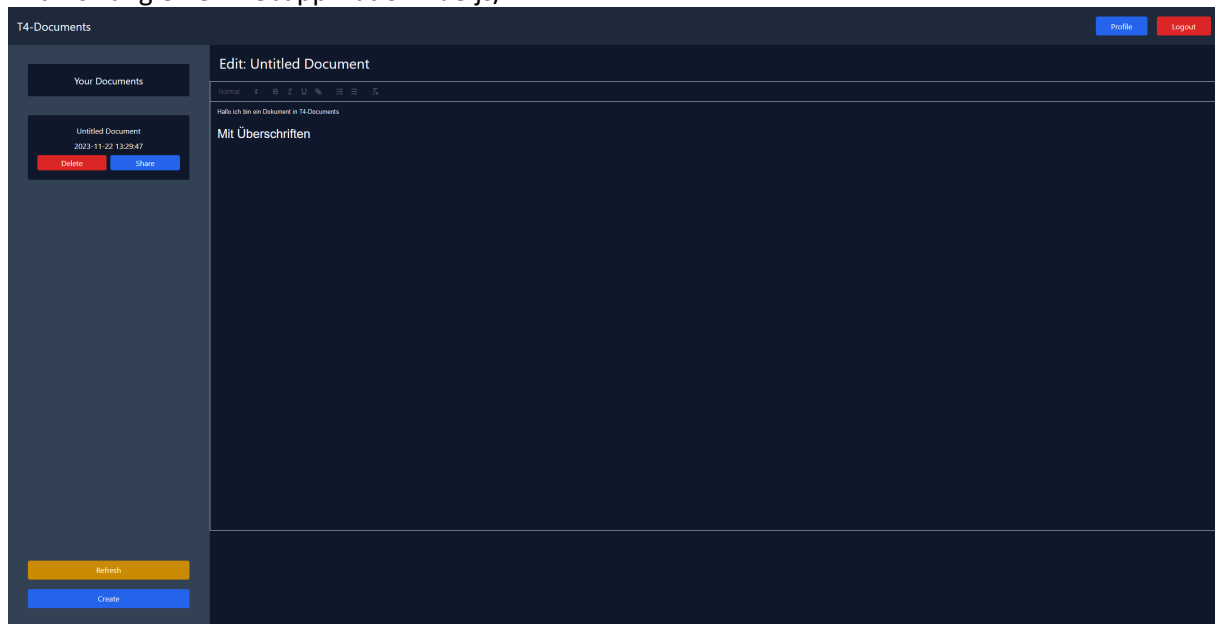


Das Ergebnis des Projekts ist eine Anwendung, welche zunächst den Login des Nutzers abfragt bzw. es dem Nutzer erlaubt, sich zu registrieren. Hier kann man Daten angeben, die man später im Profil sehen und zum Einloggen nutzen möchte.

Nun ist man auf dem Main-Screen.

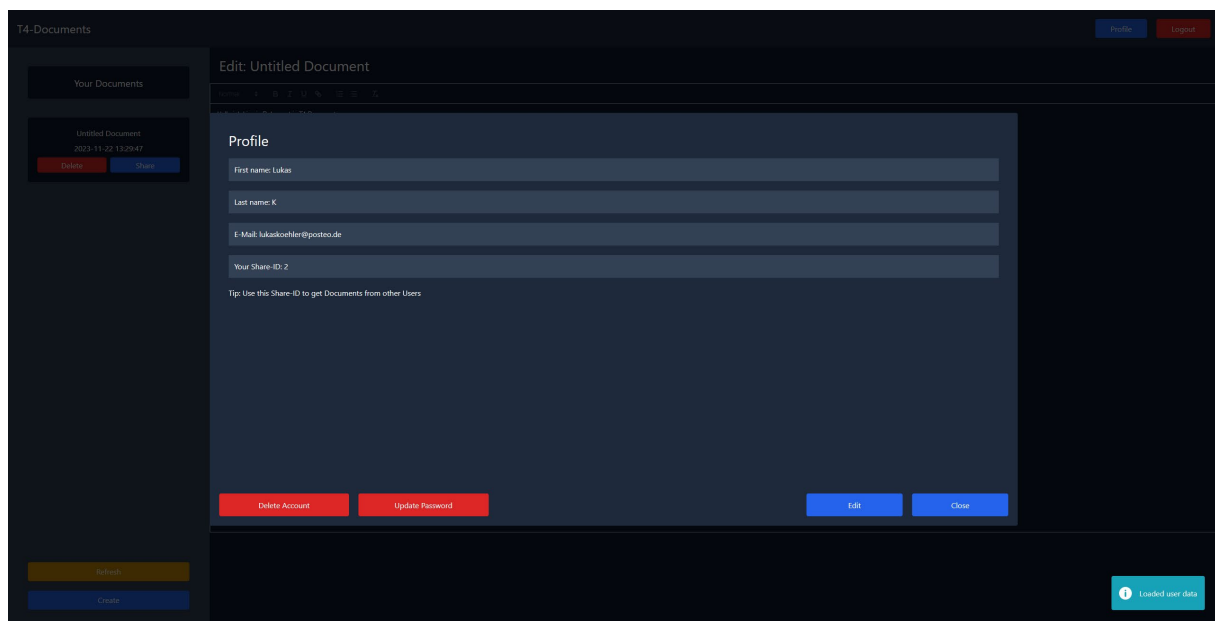


Auf der linken Seite können Dokumente ausgewählt werden, unten befinden sich Buttons, um ein neues Dokument zu erstellen oder auch um die Liste der Dokumente neu zu laden (falls jemand gerade ein Dokument geteilt hat). Mithilfe eines grünen Indikators kann man erkennen, wenn ein Dokument mit Ihnen geteilt wurde. Ohne diesen Indikator ist das Dokument in Ihrem Besitz.

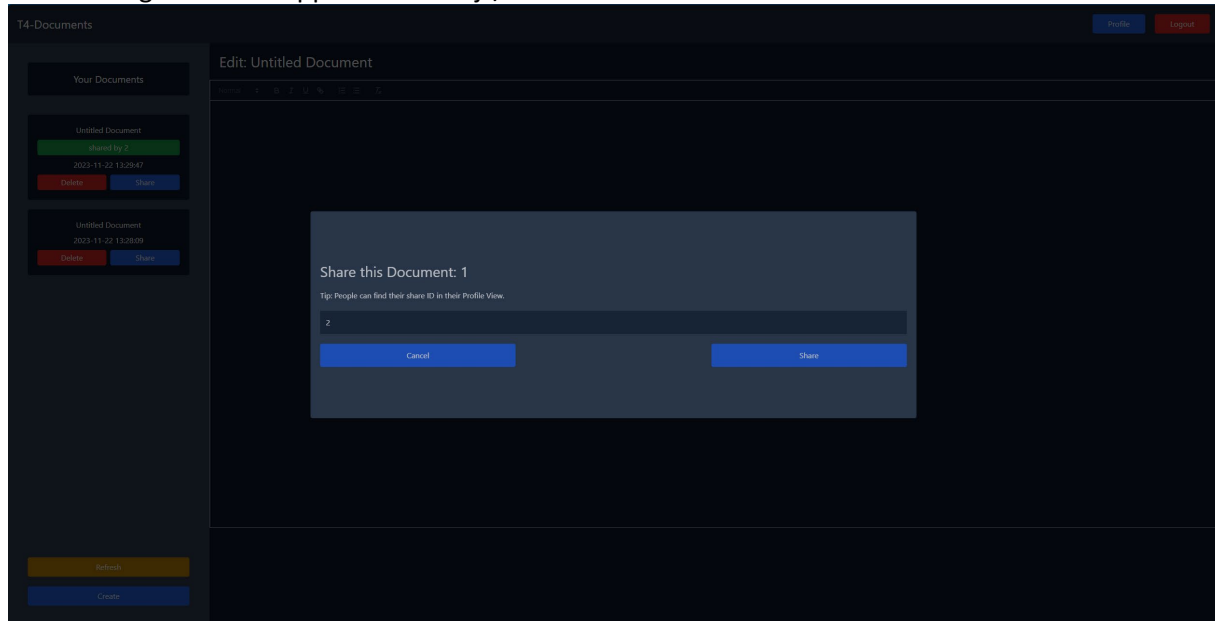


Mit dem Auswählen eines Dokuments meldet die Applikation dem WebSocket Server sofort, dass Sie das Dokument editieren. Änderungen werden auch bei nicht-geteilten Dokumenten mithilfe des Sockets kommuniziert, allerdings nicht auf andere Verbindungen geteilt.

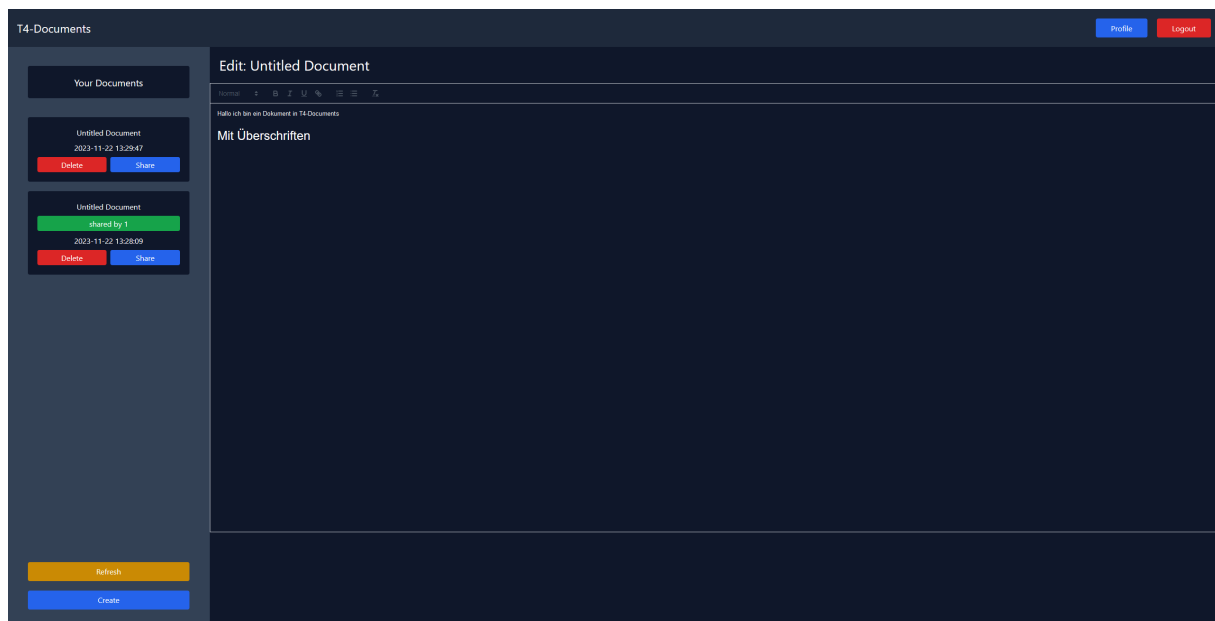
Angenommen, Sie möchten nun das Dokument einem anderen Benutzer teilen: Fragen Sie nach dessen Share-ID, diese kann im Profile View gefunden werden:



Der andere User kann nun diese Share ID nutzen, um mittels des Share Dialogs das Dokument zu teilen:

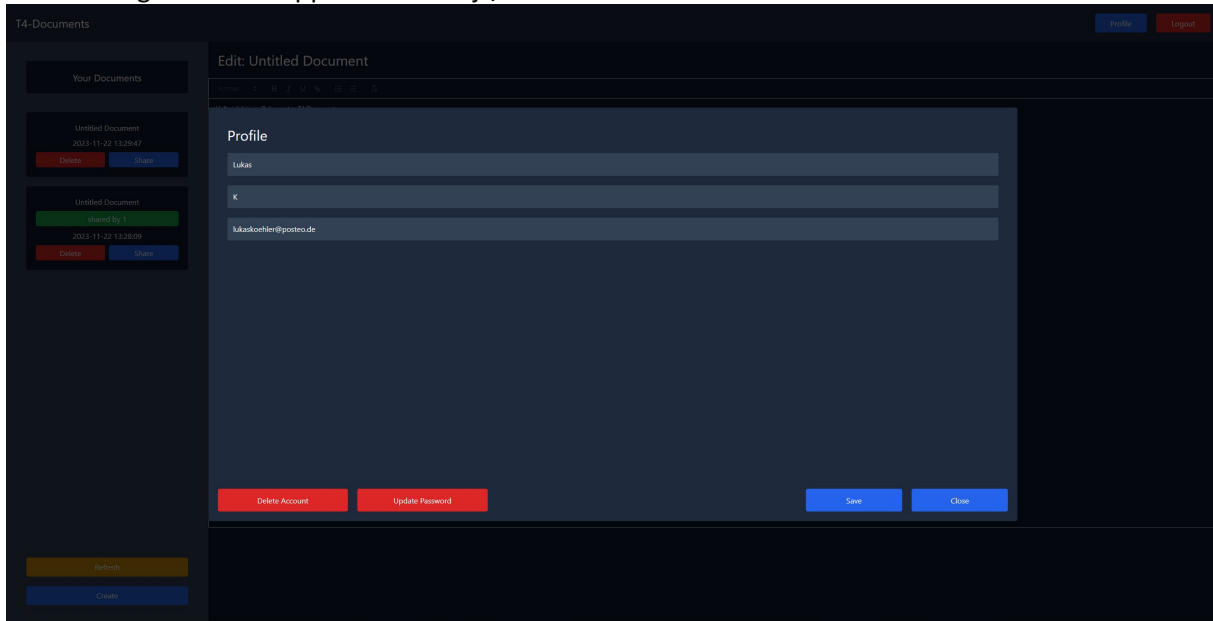


Mit Klicken des Share Buttons wird der User mit der ID als Mitbesitzer des Dokuments deklariert und kann dies nun in seiner Leiste sehen:

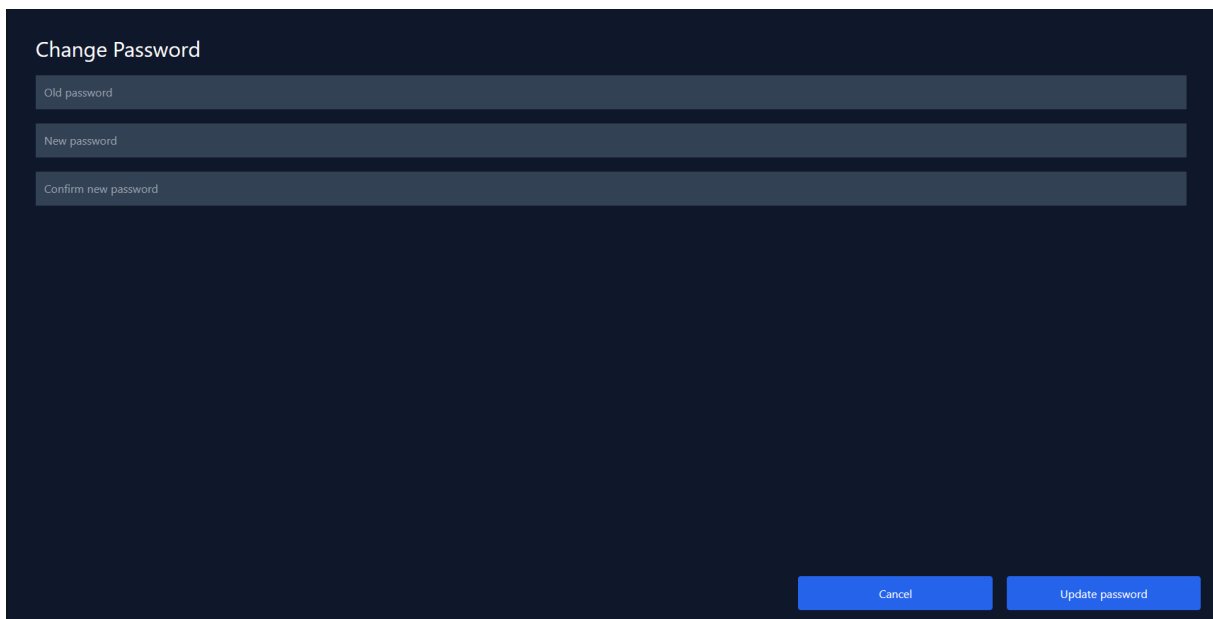


Wählt nun dieser Nutzer das Dokument aus, werden alle Änderungen synchronisiert. Alle User, die das Dokument gerade editieren, erhalten die Änderungen über den WebSocket.

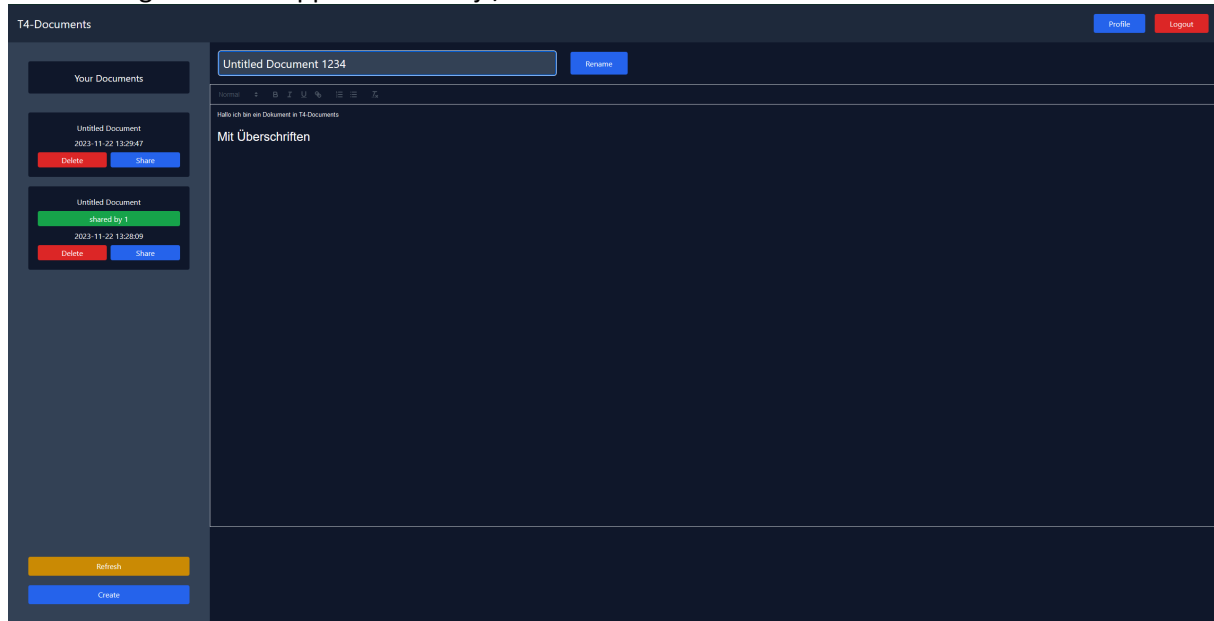
Ein weiteres Feature ist das Editieren der eigenen Nutzerdaten sowie des Passworts. Dies kann im Profile View erledigt werden:



Neue Daten eintragen, save drücken, erledigt! Sofern die E-Mail bereits belegt ist, wird eine Meldung angezeigt. Ein neues Passwort festzulegen, verläuft ähnlich:



Um ein Dokument umzubenennen, kann man einfach auf den Namen des Dokuments klicken.



Mit Klick auf „Rename“ wird der Name gesetzt.

Insgesamt bin ich mit dem Ergebnis des Projekts sehr zufrieden. Dennoch hätte ich noch einige Features, welche ich in Zukunft implementieren würde, um vor allem die UX der Anwendung zu verbessern. Aus Zeitgründen werden diese jedoch nicht mehr im Rahmen dieses Projekts behoben.


- In der „shared by“ Badge wird derzeit nur die ID des Nutzers angezeigt, besser wäre dessen Name
- Aktuell wird das Umbenennen der Dokumente nicht in Echtzeit synchronisiert. Besser wäre es, wenn der Socket auch Umbenennungen und vergleichbar handhaben kann.
- Aktuell ist die Share-ID relevant fürs Teilen von Dokumenten, in Zukunft könnten auch E-Mail-Adressen für dies relevant werden.

Dennoch konnte ich im Laufe der Zeit sehr viele Kinderkrankheiten beheben und bin vor allem stolz darauf, dass die Echtzeitsynchronisation so gut funktioniert.

7 Reflexion

7.1 Herausforderungen

Die größte Herausforderung war zunächst, den VueQuill Editor zu verstehen. Die Dokumentation war sehr kurzgehalten und enthielt kaum / keine Codebeispiele. Daher musste ich recht lange mit dem Editor kämpfen, was mir weniger Spaß gemacht hat. Dennoch habe ich dann irgendwann verstanden, wie ich Quill zu verwenden habe und konnte damit arbeiten. Besonders hilfreich war das Internet allgemein sowie insbesondere Stack Overflow. Künstliche Intelligenz war in diesem Fall maximal nützlich, um mir Tipparbeit in Form von GitHub Copilot zu sparen.

Die nächste Herausforderung war die Konzeption des WebSocket Servers. Ich hatte zunächst die Idee einen Hostprozess zu starten und von dort mehrere Instanzen für WebSocketServer zu bauen. Allerdings konnte ich, tatsächlich, durch Abwarten und Grübeln auf eine bessere Idee kommen, die nun hier umgesetzt vorliegt. Hilfreich war dazu vor allem YouTube, in welchem ich mir habe erklären lassen, wie Echtzeitkommunikation gemacht werden kann. Dabei bin ich auch über SSE gestolpert, was kurz als eine Alternative stand. Dennoch, es sind am Ende WebSockets geworden, weil WebSockets neben schnellen Leseoperationen auch schnelle Schreiboperationen bereitstellt. 

Lukas Köhler

ON22A-T4

Entwicklung einer Webapplikation Vue.js/PHP

Eine Herausforderung war, dass Vue.js in einem Produktions-Build anders funktioniert als im Development Server. Ich hatte die Situation, dass die App beinahe fertig war, ich sie als Docker-Container als Production Build bereitgestellt habe, und sie dann nicht mehr wie erwartet funktionierte.

Ich habe dieses Problem gelöst, indem jetzt im Docker Container der Development-Build läuft und nicht der Production Build. Dennoch war es sehr ärgerlich, weil es weitere Zeit in Anspruch genommen hat.

7.2 Unterstützung

Grundsätzlich, und es ist hier nicht anders, suche ich stets im Internet nach einer Lösung für meine Probleme. Dabei lese ich auch Stack Overflow Themen, welche nicht unbedingt genau mein Problem beschreiben. Diese Threads geben mir normalerweise entscheidende Hinweise darauf, wonach ich suchen muss. Zudem habe ich teils mit einem Kollegen aus Wirtschaftsinformatik an der FH Heilbronn über mögliche Strukturen Brainstorming betrieben.

Besonders wichtig ist mir eine gute und vor allem ausführliche Dokumentation für Libraries. Ich hatte bei VueQuill das Gefühl, dass die Dokumentation nutzlos war. Ich musste mir die Informationen von verschiedenen Quellen selbst zusammensuchen.

Wenn ich dann nicht mehr weiterwusste, wandte ich eine Strategie an: Einfach mal stehenlassen. Ich kümmerte mich um ein anderes Problem und am nächsten Tag fiel mir üblicherweise eine Lösung ein.

7.3 Lernerfolge / Fazit

Ich hatte mir für dieses Projekt vorgenommen, mich mit PHP Echtzeit Kommunikation auseinanderzusetzen und zu versuchen, diese möglichst genau zu verstehen. Das ging Hand in Hand mit meiner Entscheidung, so wenig Libraries wie möglich zu verwenden. Ich habe nun ein gutes Verständnis darüber erlangt, wie ich in PHP einen WebSocket Server aufsetzen kann, wie dieser Anfragen verarbeiten kann und wie ich Verbindungen mit selbigem verwalte. Ebenfalls habe ich gelernt, wie ich in PHP eine REST-API erstellen kann.

In Zukunft könnte ich dieses Wissen nutzen, um andere Applikationen zu entwickeln, welche Echtzeit Kommunikation benötigen oder davon profitieren.

A Installationsanleitung

Link zum README: [t4-document-application/README.md at main · lukkoeh/t4-document-application \(github.com\)](https://github.com/lukkoeh/t4-document-application/blob/main/README.md)

Die Voraussetzungen für die korrekte Ausführung der Applikation sind:

- Docker
- Docker-Compose
- Docker Port Mapping von Container zu Host muss sauber funktionieren

Alles ist so konzipiert, dass Sie im Repository mit einem einzigen Befehl die gesamte Applikation starten können.

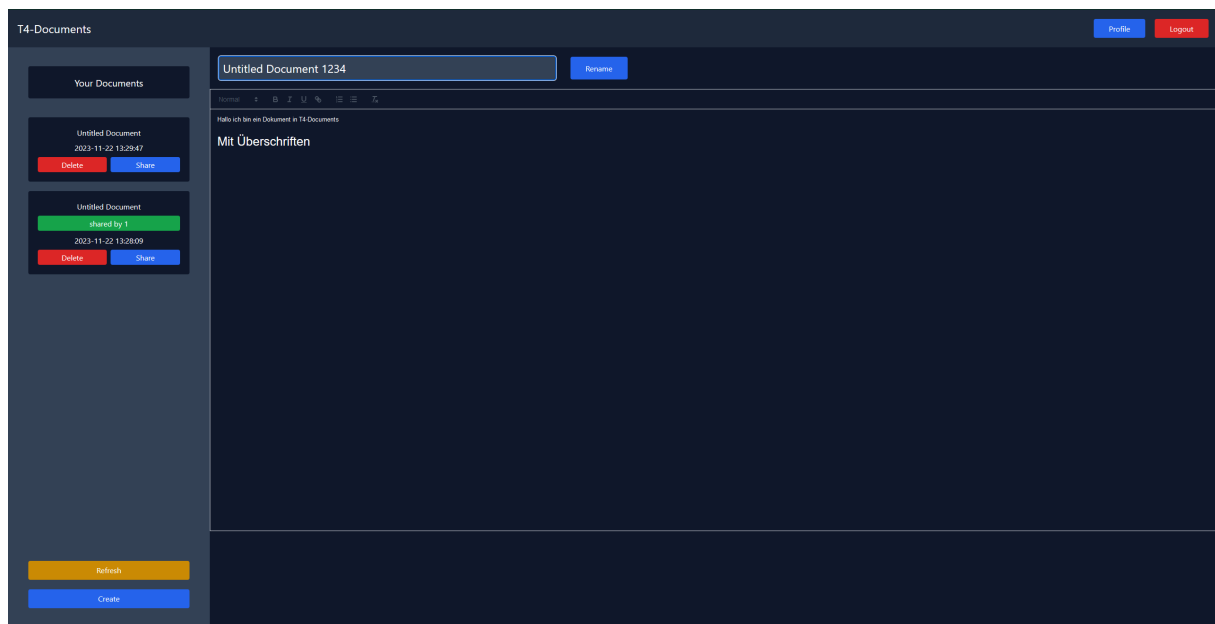
```
docker compose up -d
```

B Benutzerhandbuch

Use Case 1: Ein Dokument erstellen und teilen

Um ein Dokument zu erstellen, muss zunächst einen Account angelegt werden. Dies kann über das Registrierungsformular erledigt werden. Anschließend wird über den Button „Create“ ein Dokument angelegt sowie ein Name für selbiges festgelegt.

Sobald das Dokument angelegt wurde, muss man sich von einem anderen Benutzer die Share-ID besorgen. Diese findet der Benutzer in der Profilansicht. Mithilfe dieser ID kann der Benutzer dann das Dokument teilen. Sobald das Dokument geteilt wurde, kann der andere Benutzer es in seiner Liste sehen (Refresh-Button drücken). Sobald nun beide Benutzer das Dokument editieren, werden die Änderungen der Nutzer gegenseitig synchronisiert.



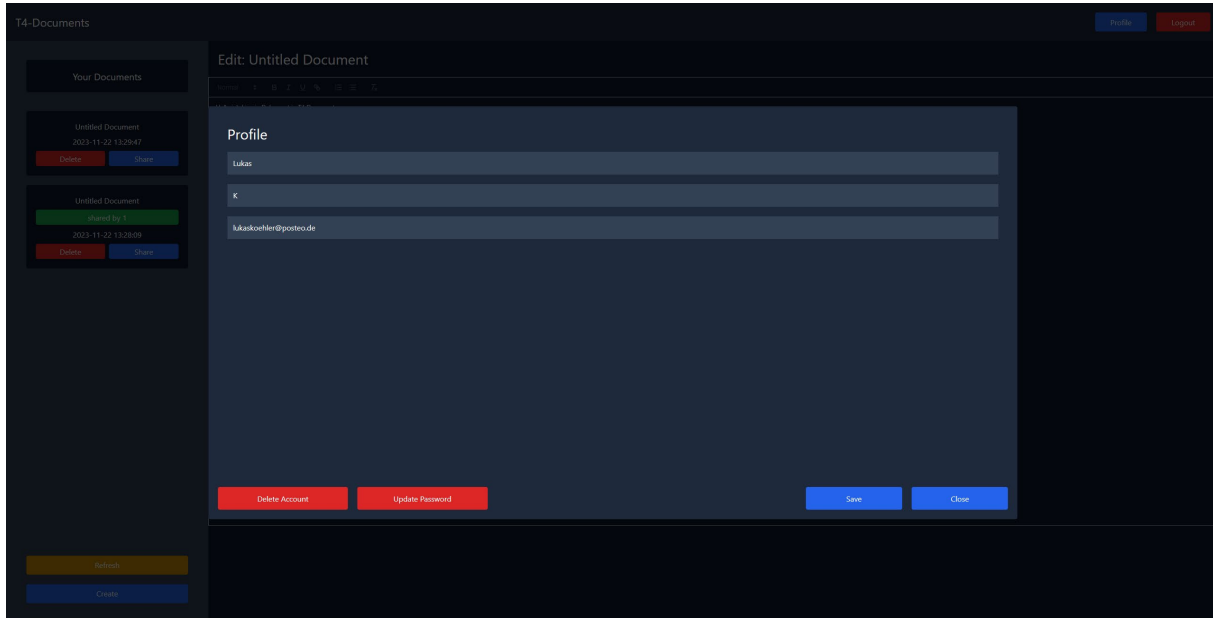
Lukas Köhler

ON22A-T4

Entwicklung einer Webapplikation Vue.js/PHP

Use Case 2: Ich möchte meine Daten ändern

In diesem Use Case möchte der Benutzer seine Benutzerdaten bearbeiten. Hierzu wechselt er nach dem Einloggen in den Profile View. Dort findet man einen Button „Edit“, den man betätigt. Nun lassen sich die Benutzerdaten beliebig editieren (abseits von bereits verwendeten Usernames). Mithilfe des Buttons „Save“ können die Nutzerdaten gespeichert werden.



Use Case 3: Ich möchte meinen Account löschen

Wenn man seinen Account löschen möchte, kann dies mit dem Profile View erledigt werden. Dort befindet sich im eingeloggten Zustand der Button „Delete Account“, welcher zunächst betätigt wird. Anschließend muss der Vorgang mit einem weiteren Klick bestätigt werden. Der Account wird nun wirklich gelöscht.

