Anwendungsentwicklung - Chat

Portfolio Anwendungsentwicklung

Studiengang

Kurs

Fakultät

Studiengang

Bearbeiter:

Betreuender Dozent:

Inhaltsverzeichnis

Seite

Abkürzungsverzeichnis

API	Application programming interface
DB	Datenbank
DTO	Data Transfer Object
ERM	Entity-Relationship-Model
REST	Representational State Transfer
URI	Uniform Resource Identifier

Abbildungsverzeichnis

Seite

Tabellenverzeichnis

Seite

Anhangsverzeichnis

Seite

1 Einführung & Ziele

Im Rahmen des x. Semesters ist die Aufgabe im Modul „Anwendungsentwicklung“ einen kleinen Chatraum zu programmieren. In diesem soll es möglich sein, dass zwei Personen mit einander kommunizieren können. Die spezifisch geforderten Funktionalitäten des Chats sind, das mehrere Clients in einem lokalen Netzwerk oder mit einer lokalen Maschine miteinander kommunizieren.

In dieser Dokumentation wird erklärt, welche Komponenten es in einem solchen Server-Client orientierten System gibt und wie diese zusammenarbeiten, um die Funktionalitäten des Programms erfolgreich zu implementieren. Zusätzlich soll die Frage geklärt werden, welche Typen von Nachrichten im System versendet werden und wie die Verarbeitung der Antworten auf diese Nachrichten erfolgen soll. In Kapitel zwei werden die Qualitätsziele für die Architektur und die Architektur selbst fixiert, bevor Kapitel drei die technische Umsetzung beschreibt. Eine kurze Rezension dieser Gruppenarbeit wird diese Dokumentation abschließen.

2 Qualitätsziele für die Architektur

Im Rahmen der Ausarbeitung haben sich vier wesentliche Qualitätsziele für die Architektur herausgestellt. Die Benutzbarkeit, die Chatanwendung sollte sowohl von mehreren Clients auf einer lokalen Maschine als auch von mehreren Clients in einem lokalen Netzwerk nutzbar sein. Die Lösung hierfür ist eine Client-Server Architektur, die keine monolithische Anwendung ist. Das bedeutet dass während das Frontend beim Nutzer liegt, dieses genutzt wird, um mit dem entfernten Backend zu kommunizieren, welches die Geschäftslogik und Datenhaltung beinhaltet. Das zweite Qualitätsziel ist, eine möglichst hohe Leistungseffizienz des Programmes zu erreichen. Die Auslieferung einer Nachricht an den Empfänger soll sofort erfolgen. Die Latenzzeit darf höchstens eine Sekunde betragen. Als Lösungsansatz wurde hier das Minimieren der Verarbeitung der Nachrichten festgelegt. Zusätzlich wird die Latenz getestet, um sicher in der Anforderungszeit zu liegen. Ein weiteres Ziel ist die Schnittstellenspezifikation zwischen dem Backend und dem Frontend.

Während die Kommunikation zwischen einem Client und dem Server über eine RESTful API abgewickelt wird, ist für den Kommunikationsweg von mehreren Clients zum Server und zurück ein WebSocket vorgesehen. Das vierte Qualitätsziel ist die zuverlässige Ansprache des Servers und die Datenintegrität. Die Nachrichten müssen trotz der asynchronen Kommunikationsform zuverlässig versendet und empfangen werden. Um dies sicherzustellen werden die Schnittstellen zum Server getestet - die Nachrichten dürfen nicht verloren gehen. Es wird also persistiert und dies getestet. Tabelle 1 Qualitätsziele der Architektur fasst die oben beschriebenen Ziele und Lösungen nochmals zusammen.

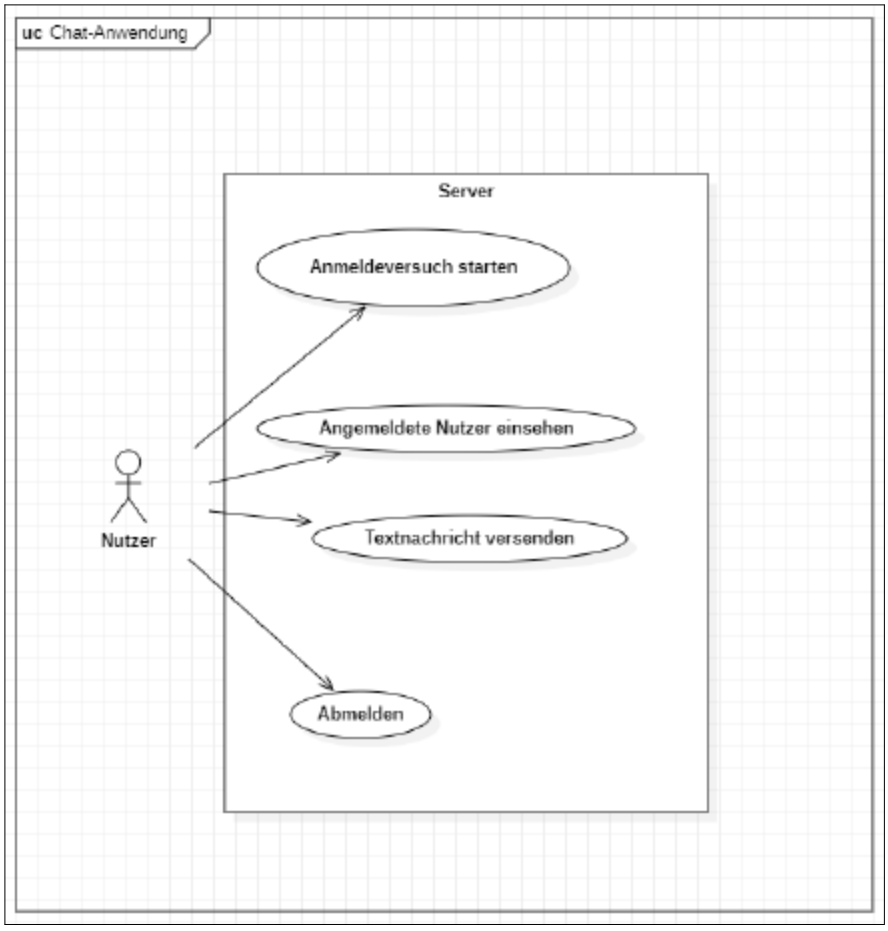
Tab. 1: Qualitätsziele der Architektur

Ziel	Beschreibung	Lösung
Benutzbarkeit	Die Chatanwendung soll sowohl von mehreren Clients auf einer lokaler Maschine, als auch von mehreren Clients in einem lokalen Netzwerk nutzbar sein.	Client-Server-Architektur keine Monolithische Anwendung
Leistungseffizienz	Auslieferung einer Nachricht an den Empfänger soll sofort erfolgen, höchstens aber mit einer Latenz von einer Sekunde	Verarbeitung der Nachricht minimieren. Latenz testen.
Schnittstellenspezifikation	Kommunikation via Schnittstelle mit Backend / Server soll spezifiziert werden.	Entwicklung einer RESTful API (Client zu Server) Entwicklung eines WebSockets (Client zu Server zu Client)
Zuverlässigkeit & Datenintegrität	Der Server muss zuverlässig ansprechbar sein. Ebenso müssen die Nachrichten trotz asynchroner Kommunikation mit dem Server zuverlässig versendet sowie empfangen werden können.	Schnittstellen zum Server testen. Nachrichten dürfen nicht verloren gehen (Persistierung + Tests).

2.1 Fachliche Perspektive

Um eine fachliche Perspektive auf das Chatprogramm zu erzeugen, zeigt Abbildung 1 ein Use Case Diagramm. Use Case Diagramme stellen und beschreiben das Systemverhalten aus Anwendersicht. Dadurch wird ein Verständnis der fachlichen Anforderungen durch die Entwickler erzeugt. Das Use Diagramm Chat-Anwendung zeigt, dass ein Nutzer mit dem Server interagiert. Dabei versucht der Nutzer u. a. einen Anmeldeversuch zu starten. Ist der Anmeldeversuch erfolgreich, kann er alle aktuell angemeldete Nutzer sehen und hat die Möglichkeit, Textnachrichten zu versenden. Nach Beendigung seiner Konversationen, kann der Nutzer sich abmelden.

Abb. 1: Use Case Diagramm Chat-Anwendung



Quelle: Eigene Darstellung

Einen Designentwurf zeigen die Anhänge II bis VI, die mithilfe der Adobe XD Software erstellt wurden.

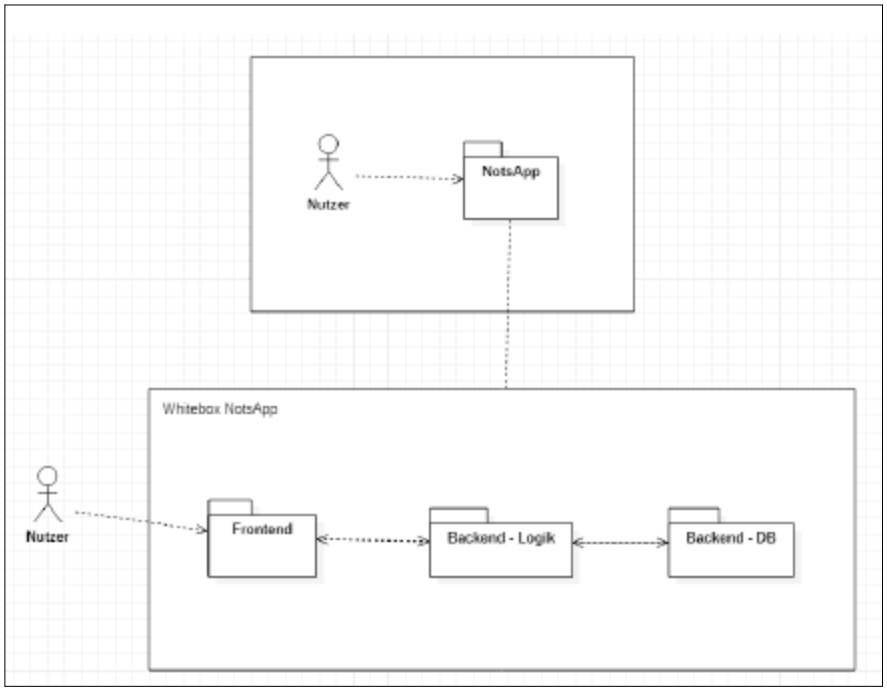
2.2 Technische Perspektive

Nun wird die Chat-Anwendung aus der technische Perspektive dargestellt.

2.2.1 Bausteinschicht

Wie in den vorherigen Kapiteln herausgearbeitet, besteht das Programm aus mehreren Bausteinen, die im Rahmen einer Microservice-Architektur miteinander verknüpft werden. Im Falle der Chat-Anwendung wird ein Frontend für die Nutzer eingerichtet und ein Backend für die logische Weiterleitung der Nachrichten. Im Frontend wird das Design der Oberfläche umgesetzt. Außerdem werden Funktionen implementiert, die dazu dienen, die Usability der Website zu erhöhen. Im Backend werden die aktuell im Chat geschriebenen Daten gespeichert, die allerdings nach Beendigung der Anwendung wieder gelöscht werden. Abbildung 2 verdeutlicht die Trennung der einzelnen Bausteine.

Abb. 2: Bausteinansicht

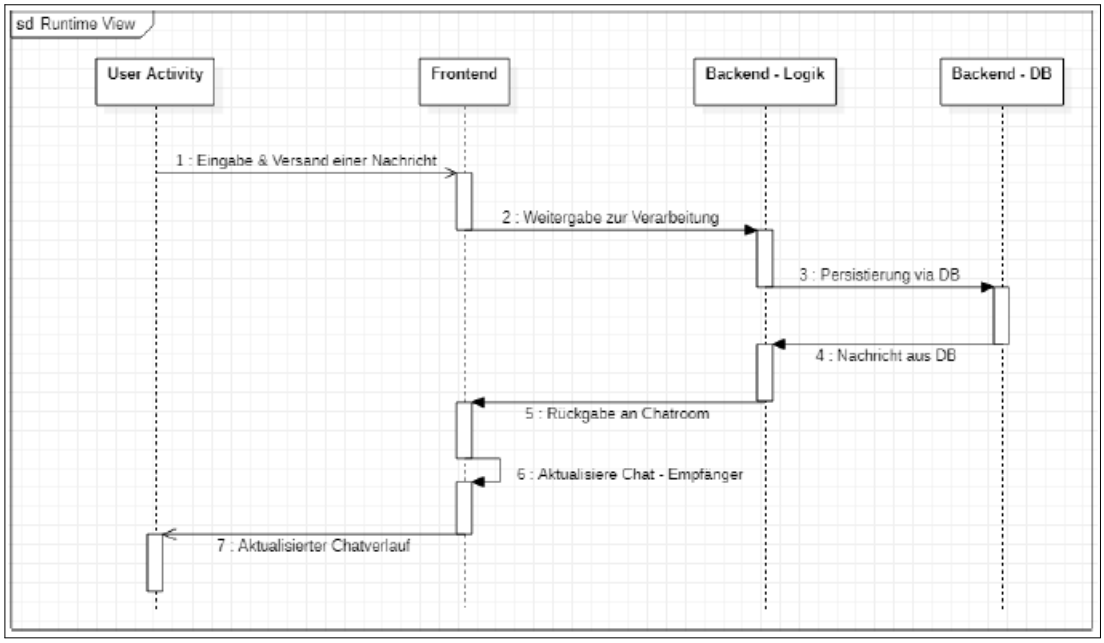


2.2.2 Laufzeitschicht

Eine Laufzeitschicht beschreibt Abläufe, die durch eine Interaktion mit dem Nutzer angestoßen werden. Abbildung 3 verdeutlicht den Hauptprozess ausgehenden vom Nutzer. Ein Chatraum wird durch eine User Aktivität mit der Eingabe und dem Versand einer Nachricht gestartet. Anschließend gibt das

Frontend die Daten an das Backend weiter. Dieses nimmt anschließend das Persistieren der Daten in der Datenbank vor. Im vierten Schritt wird die Nachricht aus der Datenbank zurückgesendet. Nun kann das Backend die Nachricht an den Chatraum zurückgeben. Das Frontend aktualisiert - sofern notwendig - den Chatraum. Der aktualisierte Chatverlauf wird dem Nutzer anzeigt. Nebenprozesse der Anwendung sind der Anmelde- und Abmeldeprozess.

Abb. 3: Laufzeitschicht



2.2.3 Verteilungsschicht

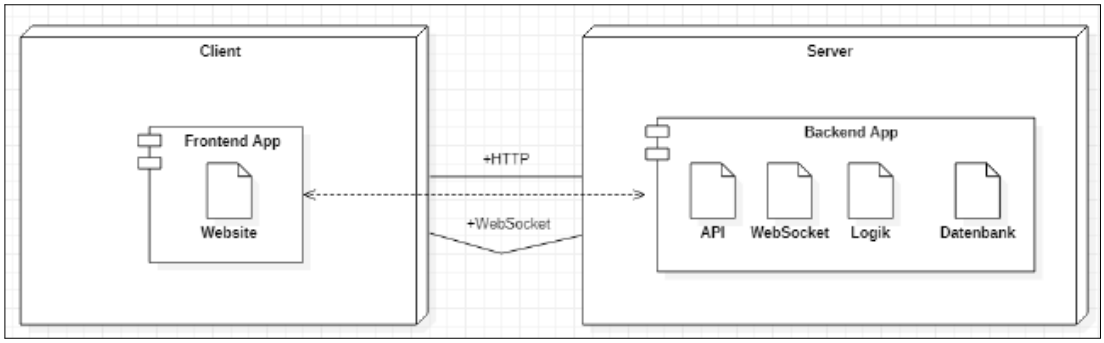
Die Verteilungsschicht zeigt einen technischen Blick auf die Konzeptionierte Anwendung. Es werden die Bausteine unter dem Aspekt ihrer zugrundeliegenden Technologien sowie deren Deployment betrachtet.

Das Web-Frontend des Chatraums wird unter Anwendung des Angular 2-Frameworks entwickelt. Es werden die Skriptsprachen HTML und CSS eingesetzt, um die die Benutzeroberfläche zu strukturieren und zu gestalten.

Zur Implementierung des Backends wurde die Programmiersprache Java mit dem Spring-Boot-Framework eingesetzt. Konzeptionell lässt sich dieser Baustein in zwei Teilbereiche gliedern. Die Schnittstelle und die Geschäftslogik.

Es existieren zwei Schnittstellen - eine RESTful API, die über HTTP kommuniziert und einen Websocket, der zum Broadcasting von Nachrichten an mehrere Clients gedacht ist. Die Geschäftslogik liegt dann dahinter und regelt die Persistierung der Daten in der Datenbank. Es handelt sich hierbei um eine in-memory Datenbank, d. h. die Daten werden zur Laufzeit geladen nach der Session wieder gelöscht. In Abbildung 4 werden die Verteilungsschichten visualisiert.

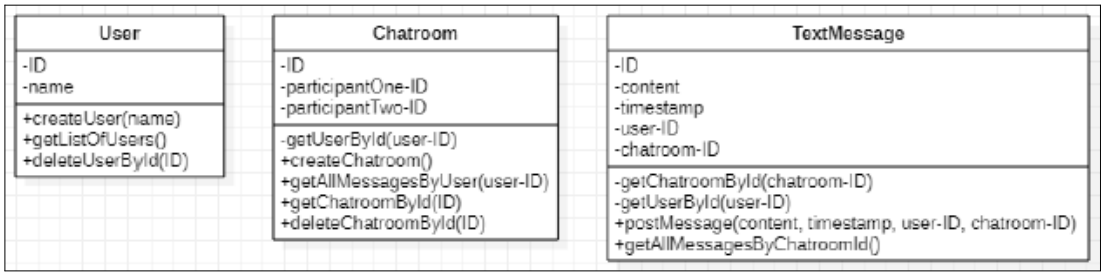
Abb. 4: Verteilungsschicht



2.3 Klassendiagramm

Das Klassendiagramm stellt die Klassen, die Vererbungsstruktur und die Beziehung zwischen Klassen dar. Abbildung 5 zeigt die Klassen User, Chatroom und TextMessage. Alle drei Klassen besitzen den Primärschlüssel ID. Die User kommunizieren jeweils in einem Chatroom. Alle Attribute sind vom Typ private, damit diese nur aus der Klasse/via Getter-Methoden aufgerufen werden können. Public-Methoden werden mittels der REST-API angesprochen. Private-Methoden hingegen sind solche, welche im BCE-Pattern der Controller zur Manipulation des Models (hier: Abruf einer Entität in der DB) dienen.

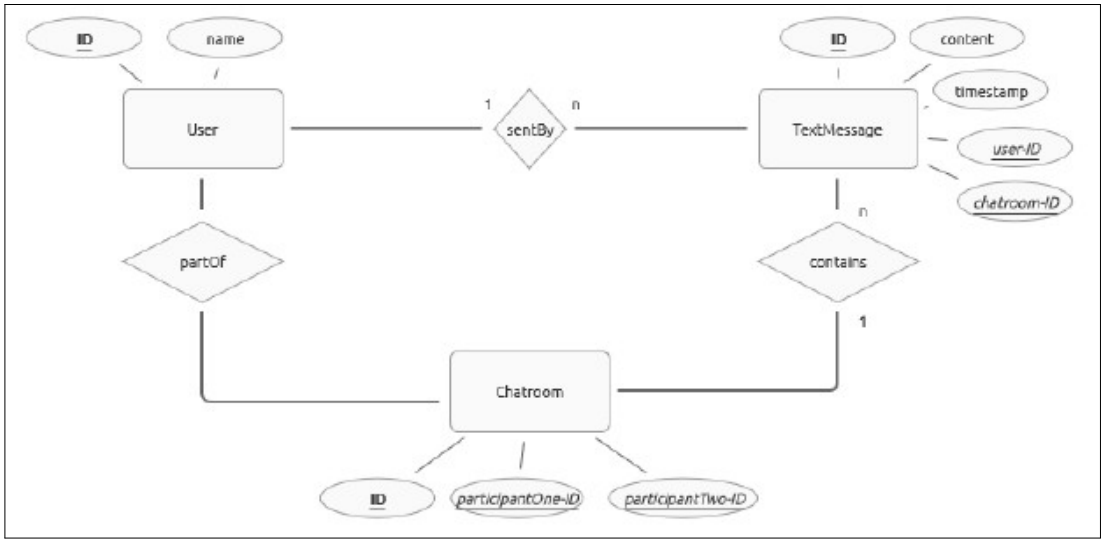
Abb. 5: Klassen



2.4 Datenbank (in-memory) – ERM

Im Backend wird zur Laufzeit eine in-memory Datenbank initialisiert, d. h. die Daten sind nach einer Session wieder gelöscht. Aus dem technisch einwandfreien ERM ergeben sich vier Tabellen (Anhang I). Wie in Kapitel 2.3 ersichtlich sind dies die Tabellen User, Chatroom und TextMessage. Außerdem ergibt sich eine Tabelle aus der n zu m Beziehung des Users mit dem Chatroom (Anhang I). Um eine anforderungsadäquate Lösung zu entwickeln wird wie gefordert nur eine Kommunikation zwischen zwei Clients implementiert und die Beziehung zwischen dem User und dem Chatroom wird auf eine 1 zu n Beziehung heruntergebrochen. Dadurch wird zusätzlich eine Performanceverbesserung erreicht, da auf weniger Tabellen zugegriffen werden muss. Abbildung 6 zeigt das zum Einsatz kommende ERM. Die Beziehung User zu TextMessage, ist eine 1 zu n Beziehung. Das bedeutet, ein User kann beliebig viele Nachrichten senden und eine Nachricht ist genau einem User zuzuordnen. Die Kardinalität zwischen der TextMessage und dem Chatroom ist n zu 1. In einem Chatroom sind beliebig viele Nachrichten enthalten und eine Nachricht kann immer einem Chatroom zugeordnet werden.

Abb. 6: ERM



2.5 Architekturentscheidungen

Auf Grundlage der zuvor ausgearbeiteten Aspekte werden in diesem Abschnitt die Architekturentscheidungen zusammengefasst. Die ERM Beziehungen zwischen dem Chatroom und dem User wurde aufgrund der Performance Verbesserung nicht mit der technisch einwandfreien Lösung aufgelöst. Es kommt eine in-memory DB zum Einsatz, das bedeutet nach Abmeldung des Users sind alle Nachrichten wieder gelöscht. Die Nutzer selbst kommunizieren in Chatrooms, die sich mit dem WebSocket verbinden. Ein WebSocket besitzt im Gegenteil zu einer gewöhnlichen API den Vorteil, dass der Server auch mit einem oder gar mehreren Clients kommunizieren kann, ohne dass eine extra Anfrage des Clients notwendig ist. Das Frontend sortiert den Chat und entscheidet über die Aktualisierung, damit muss das Backend nicht jedes Mal den ganzen neuen Chatverlauf senden. Dies erzielt eine höhere Geschwindigkeit.

3 Technische Schulden

Um ein umfassendes Bild über das entwickelte System zu geben, werden in Kapitel 3, die technischen Schulden erfasst.

- Die Lösung beinhaltet eine sicherheitstechnische Lücke, die angesichts der Anforderungen an die Software als verkraftbar angesehen wurden. So erhält ein Client aktuell sämtliche Nachrichten die über den Websocket verschickt werden und entscheidet danach erst, ob die Nachricht für diesen Client überhaupt bedeutend ist. Das zieht nach sich, dass Clients Daten erhalten, die nicht für die bestimmt sind.
- Wie in Kapitel 2.4 beschrieben, wurde ein anforderungsadäquates ERM umgesetzt. Der Vorteil dieser Umsetzung ist, dass zusätzlich eine Performanceverbesserung erreicht werden kann.
- Das Daten-Mapping ist für den Erfolg vieler Datenprozesse von entscheidender Bedeutung. Die Speicherung von Daten im Backend wurden unter Verwendung von Java Persistence umgesetzt. Dies hat zur Folge, dass bei den eins zu n Beziehungen Listen mitgeführt werden müssen. So muss die Entität User eine Liste aller ChatRooms mitführen, an denen der Nutzer teilnimmt. Um diese Listen nicht unnötig über die Schnittstelle zu schicken, wird nicht das Objekt versendet,

sondern ein sog. Data Transfer Object (DTO). Dies erhöht die Datenkonsistenz in der Datenbank.

4 Kritische Codestellen

In diesem Kapitel werden nun detailliert die wichtigsten Codestellen der Anwendung erläutert. Dazu werden die einzelnen Bestandteile, das Backend und das Frontend kurz erläutert.

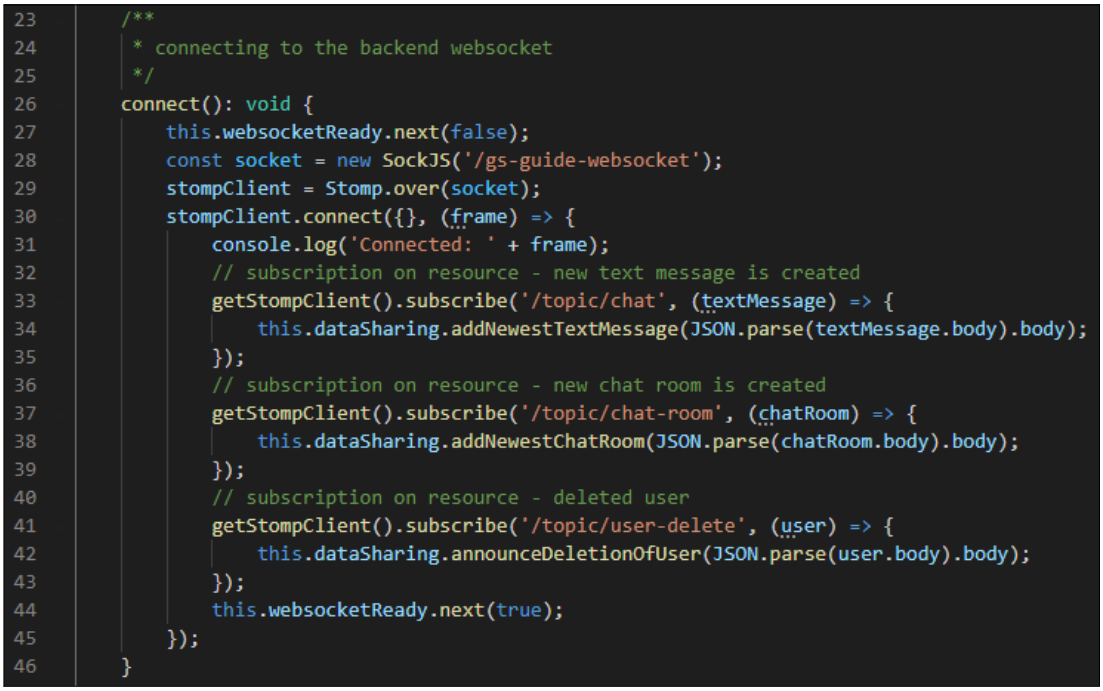
4.1 WebSocket

Durch den Websocket kann eine bidirektionale Verbindung zwischen der Webanwendung und dem WebSocket-Server hergestellt werden. Im Folgenden werden die Codestellen des WebSocket Clients und des Websocket Servers näher erläutert.

4.1.1 WebSocket Client

Die 7. Abbildung zeigt die connect()-Funktion innerhalb der Websocket-Schnittstelle am Frontend. Mit dieser Funktion wird die WebSocket-Verbindung hergestellt. Dies wird erreicht, indem die Uniform Resource Indentifiers definiert werden (URIs), auf die das Frontend hören sollen.

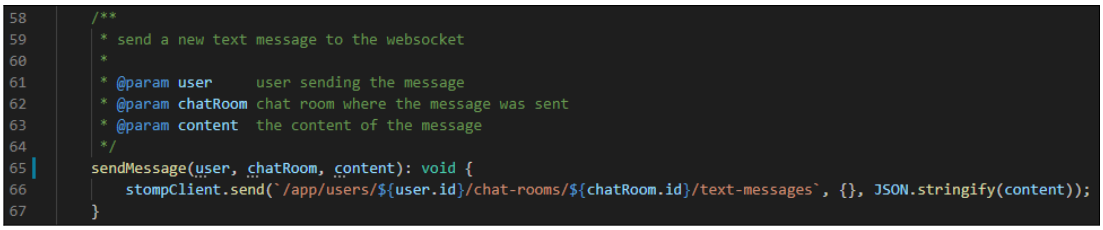
Abb. 7: Connect-Funktion



```
23      /**
24       * connecting to the backend websocket
25       */
26      connect(): void {
27          this.websocketReady.next(false);
28          const socket = new SockJS('/gs-guide-websocket');
29          stompClient = Stomp.over(socket);
30          stompClient.connect({}, (frame) => {
31              console.log('Connected: ' + frame);
32              // subscription on resource - new text message is created
33              getStompClient().subscribe('/topic/chat', (textMessage) => {
34                  this.dataSharing.addNewestTextMessage(JSON.parse(textMessage.body).body);
35              });
36              // subscription on resource - new chat room is created
37              getStompClient().subscribe('/topic/chat-room', (chatRoom) => {
38                  this.dataSharing.addNewestChatRoom(JSON.parse(chatRoom.body).body);
39              });
40              // subscription on resource - deleted user
41              getStompClient().subscribe('/topic/user-delete', (user) => {
42                  this.dataSharing.announceDeletionOfUser(JSON.parse(user.body).body);
43              });
44              this.websocketReady.next(true);
45          });
46      }
```

Abbildung 8 definiert die sendMessage-Funktion. Diese nimmt als Parameter Objekte der Klassen User, ChatRoom und den String „content“ entgegen. Dadurch kann eine Text Message an den Server gesendet werden, die anschließend an alle Clients weitergeleitet wird, die bereits eine WebSocket-Verbindung haben.

Abb. 8: sendMessage-Funktion

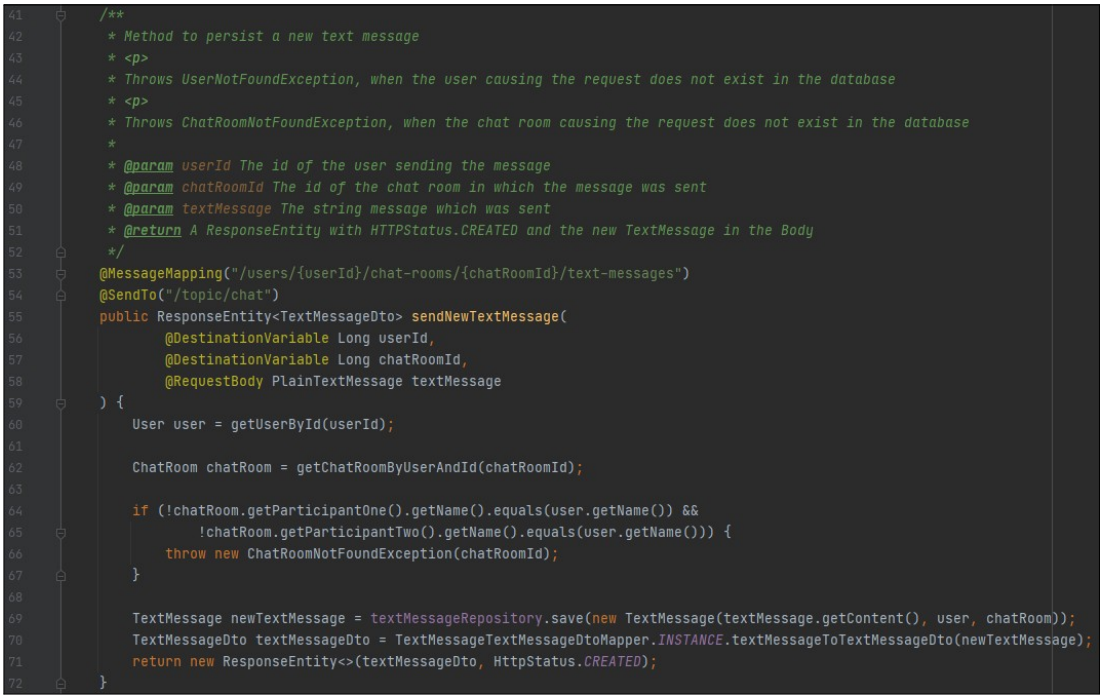


```
58      /**
59       * send a new text message to the websocket
60       *
61       * @param user     user sending the message
62       * @param chatRoom chat room where the message was sent
63       * @param content  the content of the message
64       */
65      sendMessage(user, chatRoom, content): void {
66          stompClient.send(`/app/users/${user.id}/chat-rooms/${chatRoom.id}/text-messages`, {}, JSON.stringify(content));
67      }
```

4.1.2 WebSocket Server

Die sendNewTextMessage-Methode ist die serverseitige Methode, um den Request zum Erstellen einer Text Message entgegen zu nehmen. Wichtig ist dabei die Annotation in Zeile 54, @SendTo(„/topic/chat“) diese regelt, an welche Adresse die neu persistierte Text Message gesendet werden soll, damit Clients auf diese subscriben können.

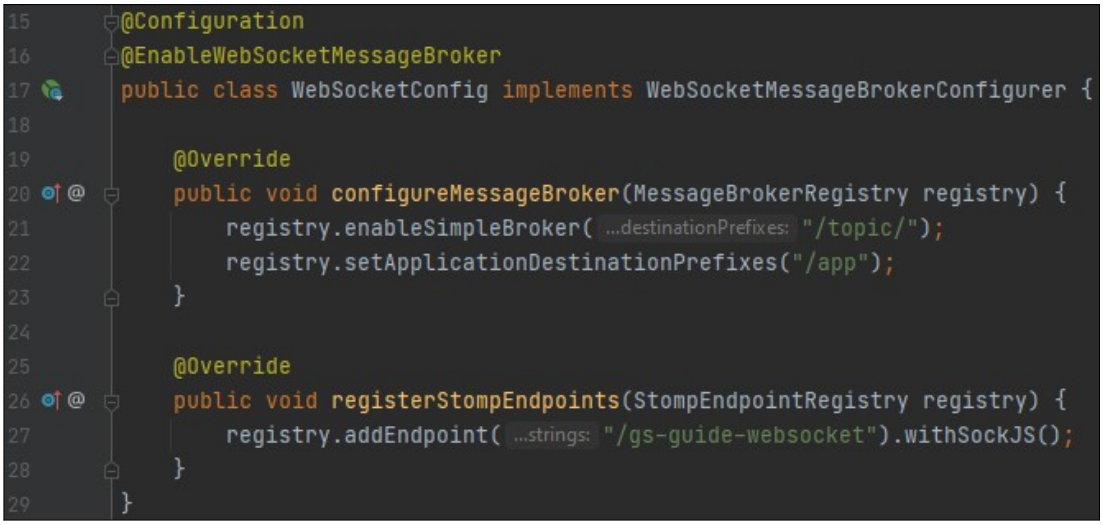
Abb. 9: sendMessage-Funktion



```java
/**
 * Method to persist a new text message
 * <p>
 * Throws UserNotFoundException, when the user causing the request does not exist in the database
 * <p>
 * Throws ChatRoomNotFoundException, when the chat room causing the request does not exist in the database
 *
 * @param userId The id of the user sending the message
 * @param chatRoomId The id of the chat room in which the message was sent
 * @param textMessage The string message which was sent
 * @return A ResponseEntity with HTTPStatus.CREATED and the new TextMessage in the Body
 */
@MessageMapping("/users/{userId}/chat-rooms/{chatRoomId}/text-messages")
@SendTo("/topic/chat")
public ResponseEntity<TextMessageDto> sendNewTextMessage(
        @DestinationVariable Long userId,
        @DestinationVariable Long chatRoomId,
        @RequestBody PlainTextMessage textMessage
) {
    User user = getUserById(userId);

    ChatRoom chatRoom = getChatRoomByUserAndId(chatRoomId);

    if (!chatRoom.getParticipantOne().getName().equals(user.getName()) &&
            !chatRoom.getParticipantTwo().getName().equals(user.getName())) {
        throw new ChatRoomNotFoundException(chatRoomId);
    }

    TextMessage newTextMessage = textMessageRepository.save(new TextMessage(textMessage.getContent(), user, chatRoom));
    TextMessageDto textMessageDto = TextMessageTextMessageDtoMapper.INSTANCE.textMessageToTextMessageDto(newTextMessage);
    return new ResponseEntity<>(textMessageDto, HttpStatus.CREATED);
}
```

In Abbildung 10 findet die Konfiguration des WebSockets statt. Hier wird festgelegt, über welche Adressen der WebSocket zu erreichen ist.

Abb. 10: WebSocket Konfiguration



```java
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker( ...destinationPrefixes: "/topic/");
        registry.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint( ...strings: "/gs-guide-websocket").withSockJS();
    }
}
```

4.2 Data Sharing-Service

Der Data Sharing-Service ist für die Konsolidierung und Zentralisierung unserer Prozesse zuständig. Die Variable observableNewestTextMessage ist eine subscribe-bare Version des BehaviorSubjects newestTextMessage. Die vari-

able newestTextMessage kann innerhalb des DataSharingService über Funktionen verändert werden. Es ändert sich dadurch auch der Wert, der hinter dem dazugehörigen Observable steckt. Auf das Observable kann man sich dann aus verschiedenen Komponenten asynchron subscriben.

Abb. 11: Variable observableNewestTextMessage

```
16    private newestTextMessage: BehaviorSubject<TextMessage> = new BehaviorSubject(null);
17    observableNewestTextMessage = this.newestTextMessage.asObservable();
```

Die Funktion addnewestTextMessage, in Abbildung 12, ist für das verändern des Wertes von newestTextMessage zuständig.

Abb. 12: Funktion addnewestTextMessage

```
58    /**
59     * Service for Sharing the newest text message between Websocket and ChatComponent.
60     *
61     * @param message the text message that was just sent
62     */
63    addNewestTextMessage(message: TextMessage): void {
64      this.newestTextMessage.next(message);
65    }
```

Abschließend wird durch die Subskription auf das observableNewestText-Message, die neue Text Message im Chatverlauf eingepflegt.

Abb. 13: Funktion addnewestTextMessage

```
43    // receiving text messages which arrive via websocket
44    this.dataSharing.observableNewestTextMessage.subscribe((textMessage: TextMessage) => {
45      // bugfix: because of trouble concerning the asynchronous communication with the dataSharingService
46      // it needs to be checked whether a message is already in the message history
47      if (this.messages.length > 0 && textMessage.id === this.messages[this.messages.length - 1].id) {
48        return;
49      }
50
51      // adding a text message to the chat and scrolling down to look at it
52      if (textMessage !== null && textMessage.chatRoom.id === this.chatRoom.id) {
53        this.messages.push(textMessage);
54
55        const msgHist = document.getElementById("msgHistory");
56        msgHist.scrollTop = msgHist.scrollHeight;
57      }
58    });
59  }
```

5 Projekttabelle

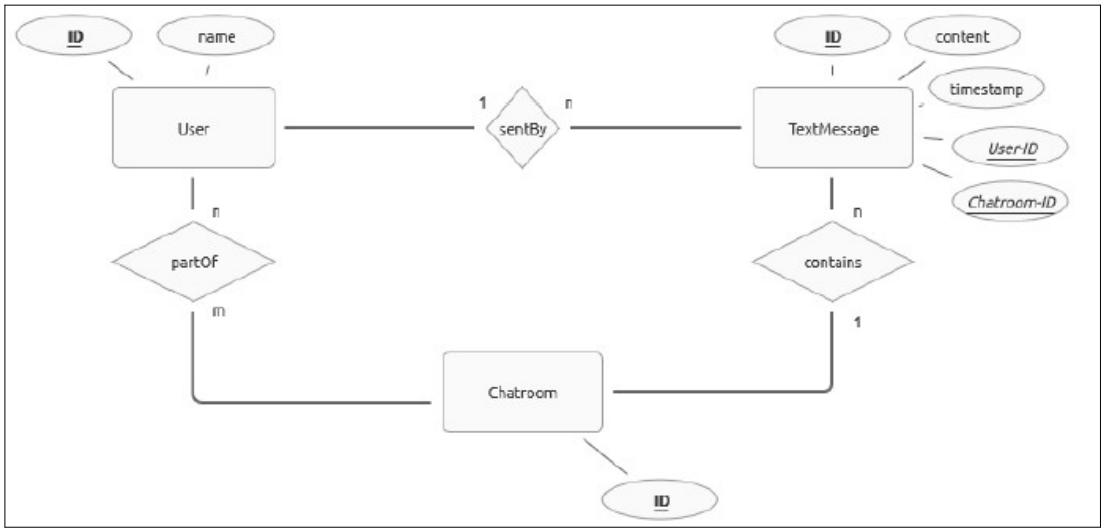
Tabelle 7 zeigt die Zuständigkeiten innerhalb der Projektumsetzung. Der Fettdruck weißt jeweils die Hauptverantwortlichkeit für diese Aufgabe aus. Es

gilt herauszustellen, dass die komplette Gruppe durch regelmäßige Teamabsprachen immer auf dem aktuellen Stand des Projektfortschritts war.

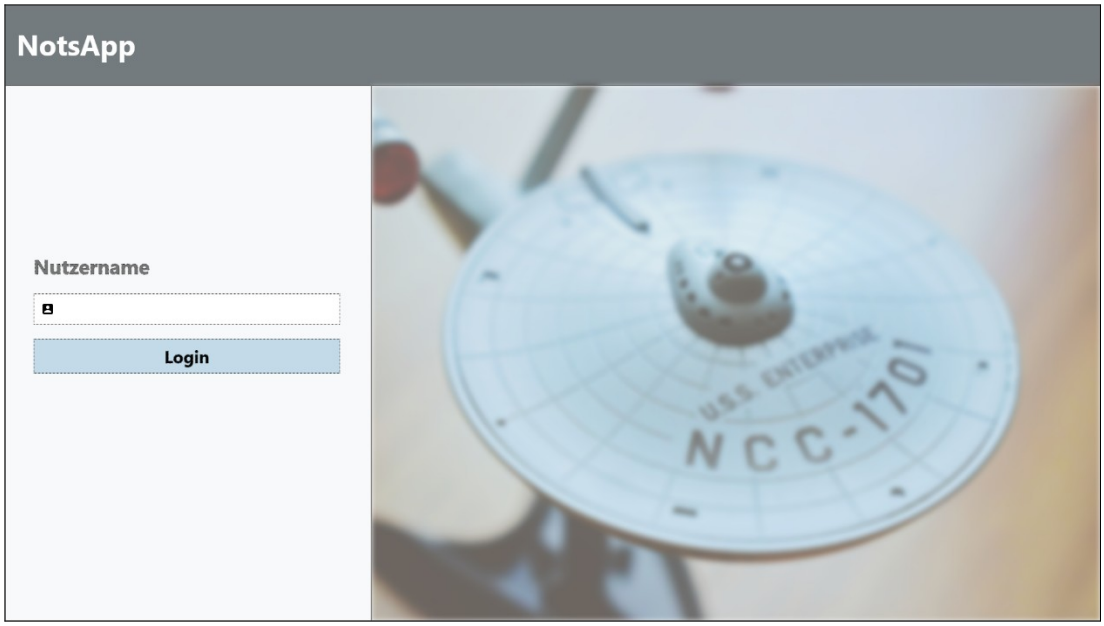
Tab. 1: Zuständigkeitstabelle

Anhang

Anhang I: Technisch einwandfreies ERM

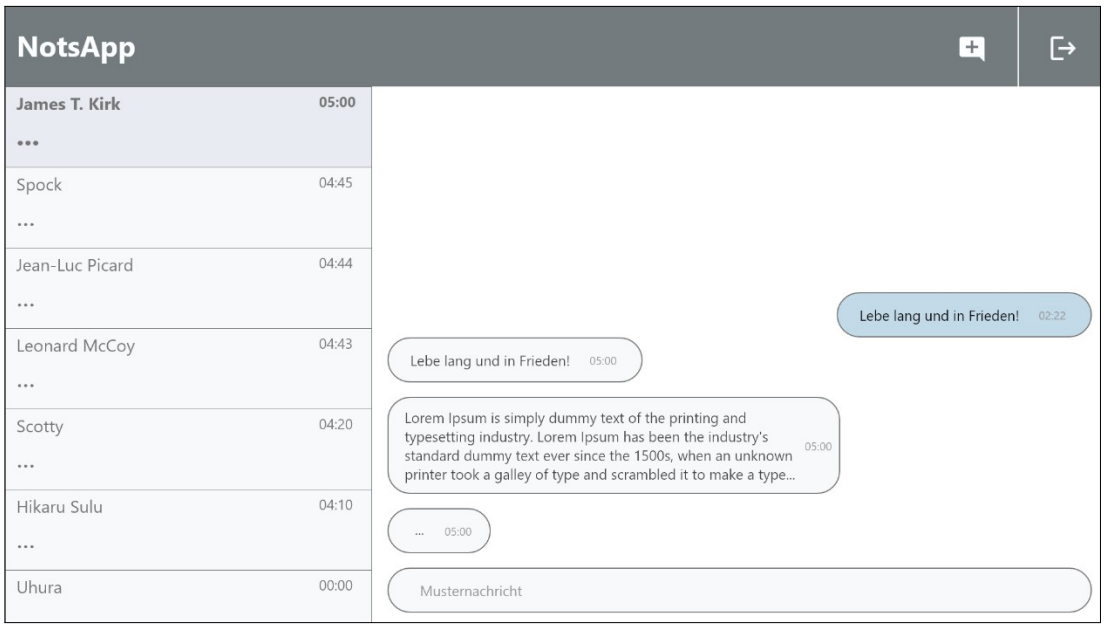


Anhang II: Adobe XD - Anmeldemaske

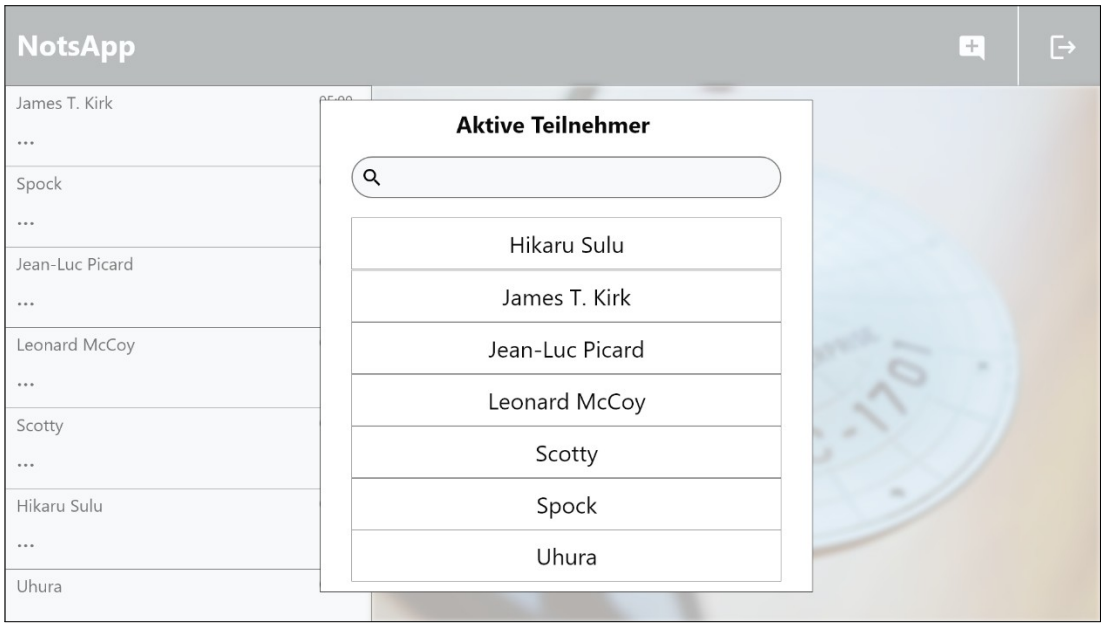


Anhang III: Adobe XD – Hauptmenü Chatroom

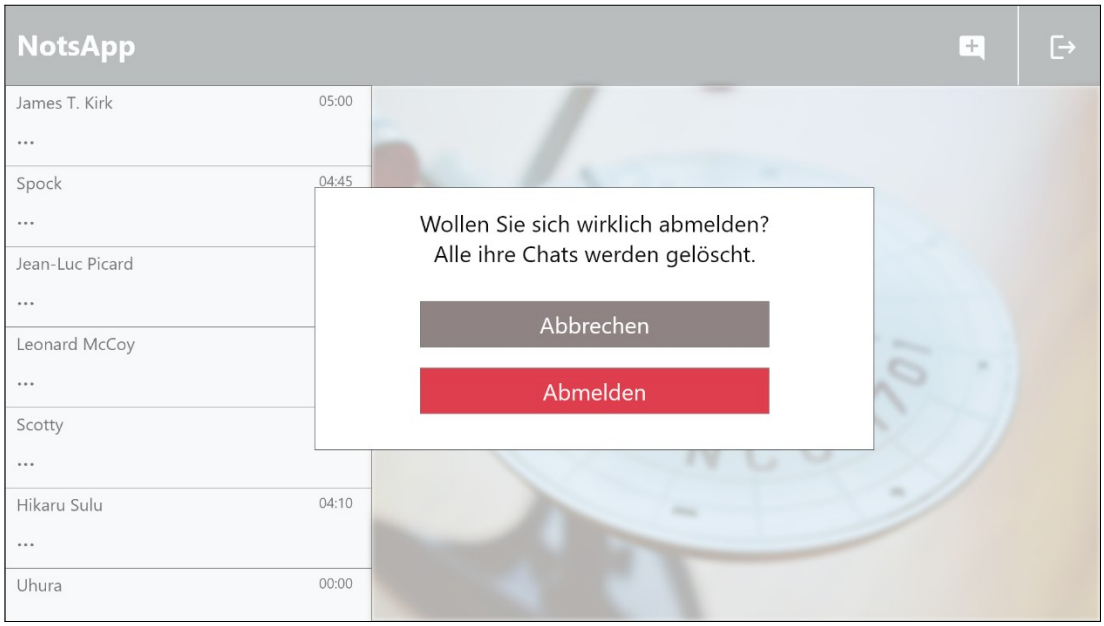
Anhang IV: Adobe XD – Hauptmenü Chatroom



Anhang V: Adobe XD – Hauptmenü Aktive Teilnehmer



Anhang VI: Adobe XD – Hauptmenü Abmelden



Erklärung

Wir versichern hiermit, dass wir unser Portfolio mit dem Thema „Anwendungsentwicklung - Chat“ selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Ort

Datum

Unterschrift