

Aufgabe 4: Chat-App Angular

In der Aufgabe 4 liegt der Schwerpunkt auf der Realisierung des Webclients als Angular Single Page Application. Mit der Fertigstellung der Aufgabe steht eine voll funktionsfähige Chat-App zur Verfügung.

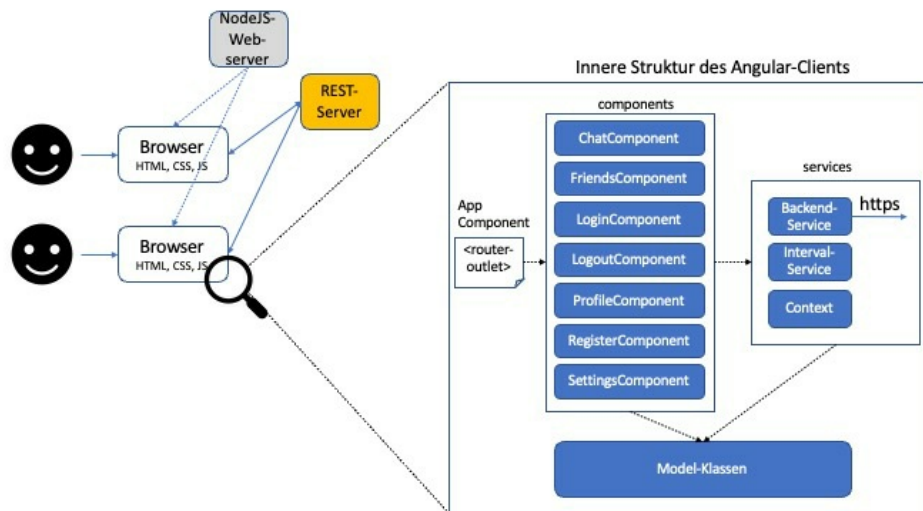
Folgende Grundlagen stellen den Ausgangspunkt für ihre Entwicklung dar:

- Die mit Aufgabenblatt 3 zu entwickelnde Anwendung, hier insbesondere die HTML- und CSS-Dateien. Die JS-Dateien werden nicht benötigt, da die Funktionalität in den Angular Components mit TypeScript umgesetzt wird. Die HTML-Dateien sind der Startpunkt für die Entwicklung der HTML-Templates der Angular Components.
- Als Datenquelle und -Senke soll der gegebene REST-Server genutzt werden. Die Dokumentation ist unter der Adresse <https://online-lectures-cs.thi.de/chat/full/56ce2af0-ee84-4e78-85bc-6bba6c51c739> zu finden.
- Als Startpunkt ist das Fragment einer Angular-App vorgegeben, die folgendes enthält:
 - eine grundlegende Software-Architektur aus "components", "models" und "services" (siehe unten)
 - Components: die HTML-Seiten aus Aufgabe 3 werden zu Components. Jede Component ist bereits vordefiniert mit einem leerem Template.
 - Models: hier findet man die Datenobjekte für die Kommunikation mit dem Server in Form vordefinierter Klassen.
 - Services: hier sind alle Service-Singletons zusammengefasst. Insbesondere ermöglicht der vordefinierte BackendService Aufrufe aller API-Endpoints des REST-Servers.

Grundsätzliche Funktionsweise und Software-Architektur der Chat-Anwendung

Die Software-Architektur beschreibt die wesentliche Struktur einer Anwendung. Die nachfolgende Abbildung zeigt diese Struktur für die Chat-Anwendung.

1. Mit *ng serve* wird die Anwendung gebaut und der NodeJS-Webserver wird gestartet.
2. Der in orange dargestellte REST-Server dient zum Lesen und Speichern der Benutzer- und Chat-Daten.
3. Öffnet man nun *localhost:4200* im Browser, so wird der gesamte Client-Code vom Webserver gelesen und im Browser zur Ausführung gebracht. Der leere Pfad wird über die Routing-Tabelle in Datei *app-routing.module.ts* automatisch auf */login* umgeleitet.
4. Das HTML-Template der Haupt-Component "AppComponent" enthält lediglich `<router-outlet></router-outlet>`, entsprechend wird gemäß der Routing-Tabelle die LoginComponent angezeigt.



5. Nun zur Grundstruktur des Angular-Clients:

- Der Code der Components befindet sich im gleichnamigen Verzeichnis *components*. Jede Component besteht aus einem HTML-Template für die Generierung der HTML-Darstellung (*view*), einer CSS-Datei zum Styling der View, sowie einer TypeScript-Klasse für die Daten sowie die Ereignisbehandlung und sonstige Berechnungen.
- Von allen Components genutzte Dienste befinden sich in Form von Service-Klassen im gleichnamigen Verzeichnis *services*. Jeder Service wird vom System nur genau einmal instantiiert und in den Components durch *dependency injection* im Konstruktor der Component eingebunden. Hervorzuheben ist hier der *BackendService*, der Methoden zum Aufruf des REST-Servers bereitstellt.
- Components und Services nutzen Datenobjekte unterschiedlicher Ausprägung. Diese Datenobjekte sind in Form von Model-Klassen im Verzeichnis *models* zu finden.

Teilaufgabe a: Vorbereitung der Entwicklungsumgebung

Ziel dieser ersten Teilaufgabe ist es, eine auf Visual Studio Code und NodeJS basierende Umgebung für den Code-Build-Run-Zyklus aufzusetzen.

Installation von nodeJS und npm

Installieren Sie beides von <https://nodejs.org/en/download/>.

Installation von Angular-CLI und der benötigten Module

Entpacken Sie das Client-Archiv (siehe Moodle-Kurs) in ein Verzeichnis *chat-app* (Beispielname!) und öffnen Sie das Verzeichnis mit Visual Studio Code („Datei > Ordner öffnen“ im VSC-Menü). Öffnen Sie nun ein Terminal-Fenster in ihrem Client-Installationsverzeichnis (Menüpunkt „Terminal > New Terminal“) und starten Sie folgende Kommandos

```
npm install -g @angular/cli@latest
```

Daraufhin wird das Command Line Interface von Angular global installiert.

```
npm install --save-dev @angular/cli@latest
```

Dies installiert das Angular CLI lokal in ihrem Projekt.

```
npm install
```

Dies installiert die in „package.json“ genannten Module.

Erster Test der Anwendung

Führen Sie nun das Kommando `ng serve` im Verzeichnis *chat-app* aus. Die Anwendung wird im *developer mode* gebaut und der NodeJS-Webserver wird gestartet. Sie können nun die Adresse `http://localhost:4200` im Browser öffnen, um die erste Seite der Anwendung zu sehen. Sie sollten die Meldung **LoginComponent works!** sehen.

Initialisierung der HTML-Templates der Components

Kopieren Sie nun jeweils den HTML-Code unter dem `body`-Tag jeder Seite in das entsprechende Template, also z.B. aus *chat.html* in *chat.component.html*. Damit initialisieren Sie jede Component View mit einem statischen Inhalt. Später werden entsprechende *model bindings* und *event handler* ergänzt.

Im Browser sollten Sie nun eine statische Version ihrer Login-Seite aus Aufgabenblatt 3 sehen.

Initialisierung der CSS-Dateien

Kopieren Sie nun ihre CSS-Regeln aus Aufgabe 3 in die globale Style-Datei *src/styles.css*. CSS-Regeln, die nur für eine Component benötigt werden, sollten stattdessen in die CSS-Datei der Component kopiert werden.

Teilaufgabe b: Einarbeitung in die Software

Nach dem Aufsetzen der Entwicklungsumgebung und der Installation des gegebenen Codefragments ist es nun an der Zeit, ein Verständnis der vorgegebenen Software zu erarbeiten. Neben der bereits erklärten Grundstruktur sind hier die Model-Klassen und die Services zu betrachten.

Model-Klassen

Jede Model-Klasse fasst die Attribute einer fachlichen Entität zusammen, siehe zum Beispiel die Klasse *User* in *src/app/models/User.ts*. Ein User-Objekt wird vom REST-Server geliefert und hat die Attribute *username*, *friends* und *requests*, wobei letztere String-Arrays mit den Benutzernamen sind. Später kommen dann noch Profil-Informationen dazu, für die es aber die separate Klasse *Profile* gibt.

Neben *User* und *Profile* gibt es noch die Klassen *Friend* und *Message*. *Friend* bildet die Einträge in der Freundesliste ab, wo zu jedem Benutzernamen die Anzahl der ungelesenen Nachrichten angezeigt wird. *Message* definiert die Attribute einer gesendeten Nachricht, also Nachrichtentext, Sender-Benutzername und Sendezeit als UNIX-Zeitstempel in Millisekunden.

Services

Die Service-Klassen werden von Angular als Singletons (nur max. eine Instanz pro Klasse) realisiert und können so gemeinsam genutzte Daten und Operationen für die Components und Services bereitstellen.

Context

Der Service *Context* stellt die Benutzernamen des eingeloggten Benutzers bzw. des Benutzers, mit dem gerade ein Chat aktiv ist, zur Verfügung. Diese Informationen werden Component-übergreifend benötigt, der Context erspart das mühsame Übergeben der Informationen von einer Component an andere Components.

IntervalService

Die Components *FriendComponent* (Verwaltung der Freundesliste) und *ChatComponent* (Chat mit Freund:in) sollten die dargestellten Informationen (FriendComponent: Anzahl ungelesener Nachrichten je Freund:in, ChatComponent: Nachrichtenliste) zyklisch aktualisieren. Hierfür bietet sich der JavaScript Interval-Mechanismus an, mit dessen Hilfe man eine selbstdefinierte Funktion mit einer gewissen Periode im Hintergrund ausführen lassen kann. Die Funktion *id = setInterval()* setzt eine derartige Hintergrund-Aktivität, *clearInterval(id)* hebt sie wieder auf.

Der Service stellt mit *setInterval(componentName, lambda)* eine Methode zur Verfügung, die die übergebene anonyme Funktion *lambda* alle 2 Sekunden (s. *intervalTime*) ausführt. In einem *Map* wird die IntervallId zur Component gespeichert, um einerseits zu vermeiden, dass mehrere Interval-Timer gestartet werden und um andererseits mit *clearIntervals()* alle Timer löschen zu können.

BackendService

Der BackendService stellt Methoden zur Verfügung, die die API des REST-Servers aufrufen. Die API ist unter der Adresse <https://online-lectures-cs.thi.de/chat/full/56ce2af0-ee84-4e78-85bc-6bba6c51c739> zu finden.

Wichtig:

Setzen Sie in Zeile 14 die benötigte Server-Id ein!

Jede Servicemethode liefert ein *Promise<T>*-Objekt zurück, also nicht einen Wert vom Typ *T*. Das erwartete Ergebnis liegt noch nicht sofort vor, da zuerst auf die Antwort des Servers gewartet werden muss. Mit der *then*-Methode registriert man daher eine eigene Handler-Methode, die durch den Promise aufgerufen wird, sobald das Ergebnis (oder ein Fehler) vorliegt.

Die vorliegende Service-Implementierung differenziert der Einfachheit halber nicht bei Fehlern (etwa "Server nicht erreichbar", "falsche Parameter", "Aufruf nicht erlaubt"), d.h. der Aufrufer erhält *false* (bei *T == boolean*), *null* (bei *T == <Model-Typ>*, z.B. *User*) bzw. *[]* (bei *T == Array*, z.B. *Array<Message>*) zurück.

Die Nutzung der API soll beispielhaft am Loginvorgang in der *LoginComponent* gezeigt werden. Nach der Eingabe von Benutzername und Passwort stehen die beiden Werte aufgrund eines definierten *model bindings* in den Attributen *username* und *password* zur Verfügung. Wird nun der Login-Button gedrückt, so kann der Server im Button-Handler zur Überprüfung der Eingaben folgendermaßen aufgerufen werden:

```
this.backendService.login(this.username, this.password)
  .then((ok: boolean) => {
    if (ok) {
      console.log('login successful!');
      this.router.navigate([ '/friends' ]);
    } else {
      this.message = 'Authentication failed!';
    }
  });
```

Die *login*-Methode liefert *Promise<boolean>*, entsprechend wird ein Lambda-Ausdruck mit einem *boolean*-Parameter *ok* übergeben. Gilt *ok == true*, so war die Authentifizierung erfolgreich und es wird zur *FriendComponent* navigiert. Die hierfür notwendige Regel ist in der Routing-Tabelle (s. *app-routing.module.ts*) bereits hinterlegt. Andernfalls wird das Attribut *message* mit der anzuzeigenden Fehlermeldung belegt. Über ein *model binding* wird *message* ins DOM übertragen und zur Anzeige gebracht.

Routing

Eine *Route* ist ein relativer Pfad innerhalb der Anwendung, z.B. */login*. Der absolute Pfad wäre dann im Fall einer lokalen Entwicklung *http://localhost:4200/login*.

Über den optionalen Routingmechanismus ermöglicht Angular die Abbildung von Routes auf Components. Soll beispielsweise von der *RegisterComponent* auf die *LoginComponent* gewechselt werden, so kann dies auf zwei Arten bewerkstelligt werden:

1. im Template mit `<a [routerLink]="['/login']">Login`
2. in einer Methode der TS-Klasse mit `this.router.navigate(['/login'])` (dies setzt eine Einbindung des Router-Service im Konstruktor der TS-Klasse voraus!)

Grundlage für obige Navigationen ist eine Routing-Tabelle, welche mögliche Pfade in der Anwendung auf die Components abbildet. Die Routing-Tabelle für unsere Chat-App (s. *app-routing.module.ts*) ist nachfolgend gezeigt. Der erste Eintrag bewirkt eine Umleitung des leeren Pfades auf */login*, die übrigen bilden die Pfade auf die Components ab.

```
const routes: Routes = [  
  { path: '', redirectTo: '/login', pathMatch: 'full' },  
  { path: 'login', component: LoginComponent },  
  { path: 'register', component: RegisterComponent },  
  { path: 'logout', component: LogoutComponent },  
  { path: 'profile', component: ProfileComponent },  
  { path: 'settings', component: SettingsComponent },  
  { path: 'chat', component: ChatComponent },  
  { path: 'friends', component: FriendsComponent }  
];
```

Wie und wo auf der Browserseite wird nun die ermittelte Component tatsächlich angezeigt?

Hierfür gibt es das Tag `<router-outlet></router-outlet>`. In unserem Fall ist es im Template der Haupt-Component `app-component.html` als einziges Markup eingebunden, d.h. es wird immer nur die aktuell ermittelte Component im Browser dargestellt.

Teilaufgabe c: RegisterComponent

Die *RegisterComponent* ist die einzige Component, die ohne einen bereits angelegten Benutzer auskommt. Entsprechend ist es sinnvoll, diese Component als erste zu entwickeln.

Validierungen

Die auf dem Server zu speichernden Daten müssen konsistent und korrekt sein, was ein gut programmierter Server auch sicherstellt. Eine moderne Benutzeroberfläche prüft jedoch bereits vor einem Serveraufruf (hier: `register()`), ob die eingegebenen Werte korrekt sind. Diesen Vorgang nennt man *Validierung*.

Im vorliegenden Fall sollen folgende Validierungen umgesetzt werden:

- Prüfung, ob der Benutzername nicht bereits vergeben ist.
- Prüfung, ob der Benutzername eine Länge von mindestens 3 Zeichen hat.
- Prüfung, ob das Passwort eine Länge von mindestens 8 Zeichen hat.
- Prüfung, ob das Passwort korrekt wiederholt wurde.

Die Validierungen mit den entsprechenden Fehlermeldungen sind in den folgenden drei Abbildungen gezeigt. Die Validierungen sollen dabei sofort nach Loslassen einer Taste (*keyup-event*) durchgeführt werden, bei Fehlschlagen der Validierung soll die entsprechende Fehlermeldung sofort sichtbar werden bzw. gelöscht werden, falls die Validierung keinen Fehler entdeckt.

Hinweis zur Umsetzung im Template: Im input für den Benutzernamen kann eine Handler-Methode für das *keyup-event* folgendermaßen etabliert werden:

```
(keyup)="checkUsername()"
```

Benutzerführung: Funktionsausführung bei korrekter Eingabe

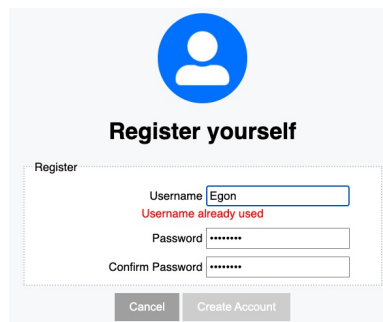
Der *Create Account"-Button sollte nur klickbar sein, falls alle Eingaben korrekt sind. Technisch bedeutet das, dass der Button ein *disabled*-Attribut besitzt, solange die Eingaben nicht korrekt sind. Das Attribut wird entfernt, wenn der Benutzer alle Eingaben korrekt vorgenommen hat.

In Angular lässt sich die oben angesprochene Funktionsweise folgendermaßen realisieren:

```
[disabled]="!usernameOk || !passwordOk">
```

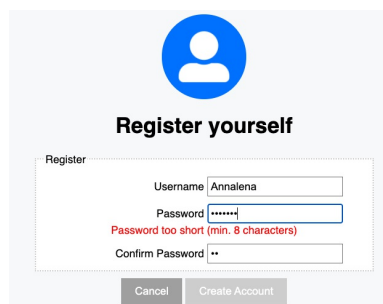
Der Button ist disabled, wenn entweder der Benutzername oder das Passwort nicht korrekt sind. *usernameOK* und *passwordOK* sind boolean-Attribute in der TS-Klasse, die durch die Validierungsfunktionen entsprechend gesetzt werden.

Im folgenden werden drei der vier Validierungen im Fehlerfall gezeigt.



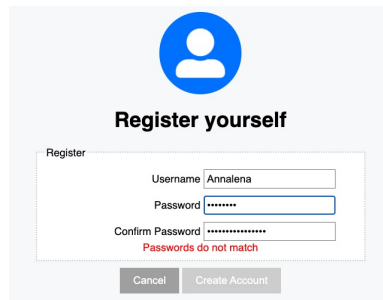
The screenshot shows a registration form titled "Register yourself" with a blue user icon. The form is labeled "Register" and contains three input fields: "Username" with the value "Egon", "Password" with masked characters "*****", and "Confirm Password" with masked characters "*****". A red error message "Username already used" is displayed below the Username field. At the bottom, there are two buttons: "Cancel" and "Create Account".

Fehlermeldung bei Eingabe eines bereits existierenden Benutzernamens.



The screenshot shows the same registration form titled "Register yourself" with a blue user icon. The form is labeled "Register" and contains three input fields: "Username" with the value "Annalena", "Password" with masked characters "*****", and "Confirm Password" with masked characters "*****". A red error message "Password too short (min. 8 characters)" is displayed below the Password field. At the bottom, there are two buttons: "Cancel" and "Create Account".

Fehlermeldung bei zu kurzem Passwort.



Fehlermeldung bei Eingabe einer nicht korrekten Passwort-Wiederholung.

Gehen Sie zur Umsetzung folgendermaßen vor (auch für alle übrigen Components empfohlen):

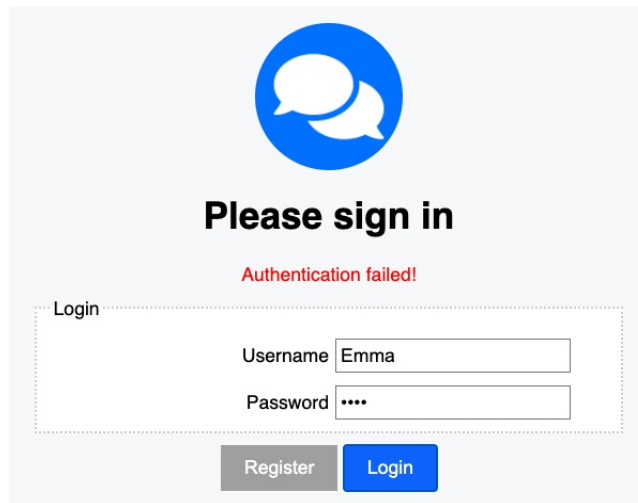
- Definieren Sie für jeden im Template anzuzeigenden Wert ein entsprechendes Attribut in der TS-Klasse. Beispiel: *public username: string = ''*; für den Benutzernamen-input
- Stellen Sie eine Model-Bindung her zwischen dem HTML-Element im Template und dem zuvor definierten Attribut. Beispiel: `<input type="text" name="username" [(ngModel)]="username" ...`
- Überlegen Sie, welche Events es in der Component gibt (z.B. Button-Klick, keyup-Event) und definieren Sie für jedes Event einen entsprechenden Event-Handler. Beispiel: *public createAccount(): void { ... }* für den Klick auf den *Create Account-Button*
- Binden Sie die Event-auslösenden Elemente an die jeweiligen Event-Handler. Beispiel: `<button type="button" (click)="createAccount()" ...`
- Rufen Sie die Component durch Eingabe des Pfades `http://localhost:4200/register` auf und testen Sie ihr Verhalten. Hilfreich sind hier Ausgaben mit `console.log('Ausgabertext...')`; bzw. das Nachvollziehen der Ausführung durch Setzen von Haltepunkten im Browser (Entwicklertools, Quellcode-Ansicht).

Hinweis für Chrome-Nutzer:

Für das Debugging mit Chrome muss in den Einstellungen des Entwicklungstools die Checkbox "enable JS source maps" aktiviert sein!

Teilaufgabe d: LoginComponent

Mit der *RegisterComponent* können wir Benutzer anlegen. Nun macht es Sinn, die *LoginComponent* zu realisieren. Folgen Sie dabei der für die *RegisterComponent* empfohlenen Vorgehensweise. Für den Aufruf der *LoginComponent* genügt aufgrund der in der Routingtabelle festgelegten Weiterleitung der Pfad `http://localhost:4200`.



Please sign in

Authentication failed!

Login

Username Emma

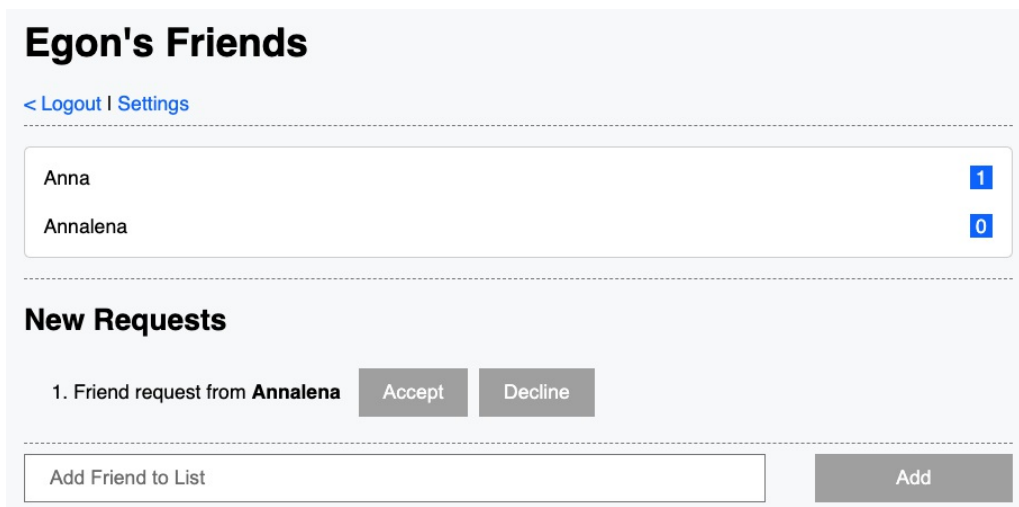
Password ****

Register Login

Schlägt die Authentifizierung durch den Server fehl, so soll eine Fehlermeldung wie oben gezeigt erscheinen.

Teilaufgabe e: FriendsComponent

Die Funktionsweise der FriendsComponent wird anhand eines Screenshots erläutert.



Egon's Friends

[< Logout](#) | [Settings](#)

Anna 1

Annalena 0

New Requests

1. Friend request from Annalena Accept Decline

Add Friend to List Add

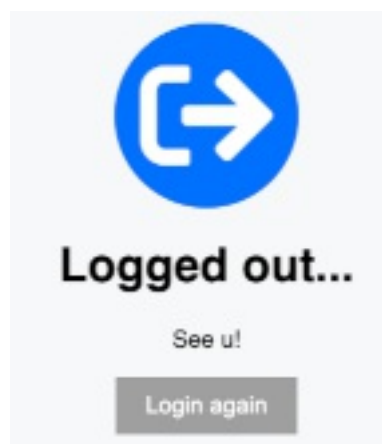
Der Dialog bietet folgende Funktionalität

- Der Logout-Link führt zur Anzeige der LogoutComponent durch Navigation zum Pfad `/logout`.
- Der Settings-Link führt zur Anzeige der SettingsComponent durch Navigation zum Pfad `/settings`.
- Die Friend-Liste stellt ein Array von Friend-Objekten dar. Diese entsteht in mehreren Schritten:
 - Zuerst muss der aktuelle User geladen werden.

- Zu den Benutzernamen in der friends-Liste erzeugt man dann *Friend*-Objekte mit initial auf "0" gesetztem Wert für *unreadMessages*.
 - Im letzten Schritte holt man sich alle *unreadMessage*-Anzahlen vom Server und setzt damit die *unreadMessage*-Werte in der Friend-Liste.
- Unter "New Requests" werden die im User-Objekt enthaltenen Benutzernamen der Freundschaftsanfragen aufgelistet.
 - Die Buttons *Accept* bzw. *Decline* werden die entsprechenden Server-Operationen aufgerufen.
 - Die oben beschriebenen Vorgänge (Friends/Requests) sollten über den *Interval/Service* periodisch wiederholt werden. Es bietet sich daher an, eine private Methode zu definieren, die dann an *setInterval()* übergeben wird.
 - Wird "Add" angestossen, so wird *backendService.friendRequest()* aufgerufen. Ist dies geschehen, sollte der angefragte Freund-Name in der Friends-Liste erscheinen.

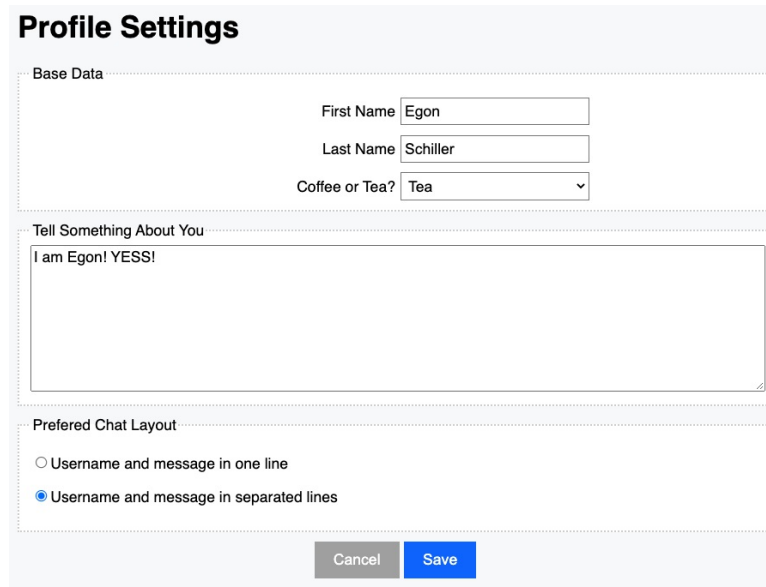
Teilaufgabe f: LogoutComponent

Die LogoutComponent ist gemäß der empfohlenen Vorgehensweise umzusetzen.



Teilaufgabe g: SettingsComponent

Mit der SettingsComponent können die Attribute eines Profile-Objekts (s. *models/Profile.ts*) belegt bzw. verändert und gespeichert werden. Hierfür sollte der aktuelle User geladen werden. Beim ersten Aufruf der Component fehlen die Profile-Attribute im User-Objekt, d.h. das zu bearbeitende Profil-Objekt (vgl. *models/Profile.ts*) sollte mit passenden Default-Werten belegt werden.



Umsetzung von "Coffee or Tea?"

Ein select-Tag kann mit Angular folgendermaßen genutzt werden, hier am Beispiel des Lieblingsvereins:

```
<select [(ngModel)]="lieblingverein" name="lieblingsverein">
  <option [ngValue]="1" [selected]="lieblingverein == '1'">
    Ich hasse Fussball!</option>
  <option [ngValue]="2" [selected]="lieblingverein == '2'">
    FC Bayern</option>
  <option [ngValue]="3" [selected]="lieblingverein == '3'">
    Borussia Dortmund</option>
</select>
```

Der gewählte Wert (1, 2 oder 3) wird im Attribut *lieblingsverein* gespeichert. Je nach initialem Wert des Attributs wird der entsprechende Eintrag vorausgewählt (*[selected]=...*). *[ngValue]* legt den Wert fest, der bei Auswahl des anzuzeigenden Texts im Attribut gespeichert wird.

Anmerkung:

Intern ist es sinnvoller, mit Zahlen (1, 2, ...) zu arbeiten, anstatt mit Strings ('Ich hasse Fussball' etc.).

Umsetzung des mehrzeiligen Eingabefeldes

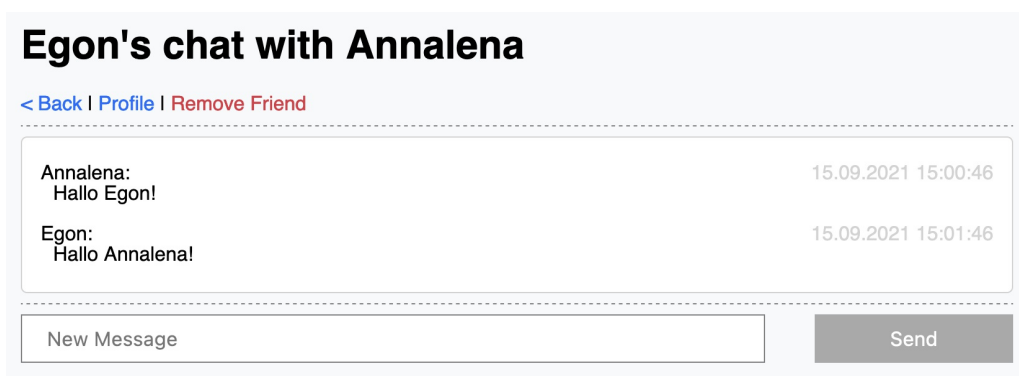
Das Modelbinding gestaltet sich bei mehrzeiligen Eingabefeldern etwas anders als bei einzeiligen. Daher ist nachfolgend ein Beispiel gezeigt, welches die Vorgehensweise veranschaulicht.

```
<textarea name="about" #about="ngModel"
  [(ngModel)]="text" row="5">
</textarea>
```

Der Wert wird aus dem Attribut *text* zur Initialisierung gelesen und bei Veränderung in das Attribut zurückgeschrieben.

Teilaufgabe h: ChatComponent

Die ChatComponent stellt die bisher ausgetauschten Nachrichten zwischen dem eingeloggten Benutzer und einer Freundin/einem Freund dar.



Egon's chat with Annalena

[< Back](#) | [Profile](#) | [Remove Friend](#)

Annalena:
Hallo Egon! 15.09.2021 15:00:46

Egon:
Hallo Annalena! 15.09.2021 15:01:46

New Message

Die Nachrichtenliste sollte periodisch aktualisiert werden, um neue Nachrichten sehen zu können. Nutzen Sie auch hierfür den `IntervalService`!

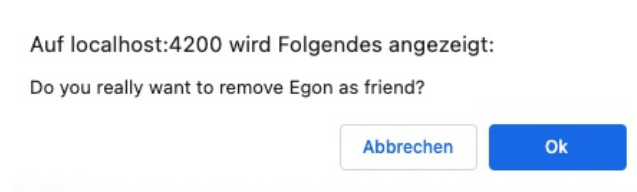
Bitte beachten:

Die Klasse `ChatComponent` enthält bereits Code, der dafür sorgt, dass die Nachrichtenliste (im Template gegeben als `DIV` mit der angular-Kennung `"#messagesDiv"`) stets an das Ende gescrollt wird. Dadurch sieht der Benutzer immer die letzten Nachrichten. Die Kommentare im Sourcecode erläutern die Funktionsweise.

Obige Darstellung zeigt die zweizeilige Darstellung von Sender und Nachricht. Ob ein- oder zweizeilig dargestellt wird, ist der jeweiligen Einstellung des Benutzers (s. *SettingsComponent*) zu entnehmen. Hierfür ist das Nutzerprofil vom Server zu laden.

Für die Darstellung des Zeitstempels in einem lesbaren Format (z.B. "15.09.2021 15:00:46") bietet Angular sogenannte *DatePipes* an, siehe auch <https://angular.io/api/common/DatePipe>.

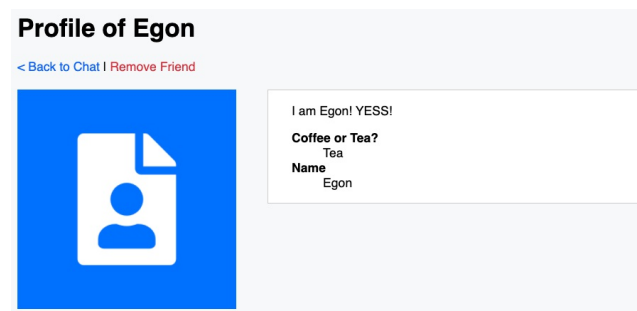
Weiterhin kann der aktuelle Chatpartner oder Chatpartnerin auch gelöscht werden (*Remove Friend*). Bevor dies geschieht, soll eine Bestätigung hierfür erfragt werden.



Nutzen Sie zur Umsetzung die JavaScript-Funktion `confirm()`.

Teilaufgabe i: ProfileComponent

Die ProfileComponent stellt Informationen über den aktuellen Chatpartner dar. Die Informationen erhält man nach dem Laden des aktuellen Benutzers aus den ggf. vorhandenen Attributen im User-Objekt. Falls das Profil (vgl. *SettingsComponent*) noch nicht gepflegt wurde, fehlen die entsprechenden Einträge.



Analog zur ChatComponent kann der aktuelle Chatpartner nach einer Bestätigung gelöscht werden.

Bewertungskriterien

- Fehlerfreie und funktionierende Umsetzung der Chat-Anwendung, dies umfasst insbesondere:
 - Anmelden mit Nutzernamen und Passwort
 - Registrieren eines neuen Nutzers mit Validierung der Eingaben und Fehlermeldungsanzeige wo nötig
 - Anzeige von Freunden
 - Hinzufügen eines Freundes
 - Annehmen oder Ablehnen von Anfragen
 - Entfernen von Freunden
 - Chatten mit einem Freund
 - Anzeige und Bearbeiten von Nutzerprofilen (nur bei dreier Teams)
- Jedes Teammitglied muss eigene Funktionsbestandteile verantworten, insbesondere Registrieren, Freundesliste und Nutzerprofil-Einstellungen sind im Team aufzuteilen