FH-TRACKING-APP

technical steps (designs/failures/solutions)

Due to failing only the exercises, I only needed to do Part 1 of the instructions. With Part 1 comes the **requirement to fake the tracking part**, but honestly it sounded interesting to me how to design such a corona-registration similar feature and **I implemented a fully functional tracking component** (room location, date and time can be picked, checked in/out and data will be saved into db + show up at overview). I used an intern array for the room-dropdown.

- [Design]: I came up with the idea to test a darkmode/lightmode feature which allows the user to switch between 2 themes via toggle.
  [Failure]: With standard CSS and the complex linking of stylesheets I was not able to fully access all styling elements via my root component, where the toggle was placed to be available at every visited component.
  [Solution]: I chose for lightmode a gradient-colored background, so the white font was visible in light- and dark mode. It only changes the background-color and font-color of the ‘h1’ tag.

- [Design]: For the project I used angular materials components in different places. Except for the ‘overview’ component, every input field is an angular material. Table, date/time-picker, themecolor-toggle and FAQ-expansion-panel are angular materials too.
  [Failure]: I struggled to use ‘matInput’ for the ‘overview’ where your basic data is shown, because calling the desired elements for styling via CSS were a pain in the a**.
  [Solution]: I used basic ‘input’ tags and style it manually until slightly satisfacted.

- [Technical]: On the first try I had some problems with calling multiple data from 2 db-objects to get for the ‘overview’ component.
  [Failure]: The table in ‘overview’ didn’t fill up with data, only showing me a row for data with no input or wrong number of rows.
  [Solution]: I forgot to adapt the correct number of arguments by calling this.http.get<{}> in the function getUserData() {}. Also calling the correct name for binding the data in overview.component.html for filling the table took me time to realize I had to use from db.js -> {{element.roomData}} like image shows below

FH-TRACKING-APP

application guidelines (how does it work)

---

component [register]:

For the registration I used the materials form input fields which get labels, formControlNames (to assign data after submitting) and mat-errors to call for validation. In the following example you can see that the first validation is the basic 'required' which applies to every input for registration. The second validation was the only elaborative one - doing a custom validator for comparing the original with the repeated password. The 'f.password_confirm' calls the function 'get f()' because we need to do this validation via controls. Instead of the keyword 'required' we call the method 'confirmPasswordValidator' which has to be built as a kind of a validation-service (own file).

```
<div>
    <mat-form-field class="example-full-width" appearance="fill">
        <mat-label>Confirm password</mat-label>
        <input matInput #input [type]="hide ? 'password' : 'text'" formControlName="password_confirm" required>
        <mat-error *ngIf="registerForm.get('password_confirm')?.errors?.required">Value required</mat-error>
        <mat-error *ngIf="f.password_confirm?.errors?.confirmPasswordValidator">Password does not match</
        mat-error>
        <button mat-icon-button matSuffix (click)="hide = !hide" [attr.aria-label]="'Hide password'" [attr.
        aria-pressed]="hide">
            <mat-icon>{{hide ? 'visibility_off' : 'visibility'}}</mat-icon>
        </button>
    </mat-form-field>
</div>
```

Equipping the inputs with validators need to be done in .html and .ts, but the custom validators get to be placed as an own object - as shown in the image below.

```
registerForm: FormGroup = new FormGroup({});
constructor(private fb: FormBuilder, private http: HttpClient, private _snackBar: MatSnackBar) {
  this.registerForm = fb.group({
    username: ['', [Validators.required]],
    password_origin: ['', [Validators.required, Validators.minLength(8)]],
    password_confirm: ['', [Validators.required]],
    email: ['', [Validators.required, Validators.email]],
    firstname: ['', [Validators.required]],
    lastname: ['', [Validators.required]]
  }, {
    validator: ConfirmPasswordValidator('password_origin', 'password_confirm')
  })
}

get f() {
  return this.registerForm.controls;
}
```

Basically the custom validator does nothing more than comparing value1 with value2, but needs to be done via formGroup controls. Don't forget to tell the validator not to fluster if everything is okay (match.setErrors(null)).

```
export function ConfirmPasswordValidator (passwBasis: string, passwMatch: string) {
    return (formGroup: FormGroup) => {
        const basis = formGroup.controls[passwBasis];
        const match = formGroup.controls[passwMatch];
        if (match.errors && !match.errors.confirmPasswordValidator) {
            return;
        }
        if (basis.value !== match.value) {
            match.setErrors({ confirmPasswordValidator: true});
        } else {
            match.setErrors(null);
        }
    }
}
```

component [login] / service [logout]:

For the html and ts part of the login component I implemented similar parts to the register component (matInputs for the html combined with requirement validation). Additionally I needed to create a loginservice to enable a user specific environment => login.

It's important to set a BehaviorSubject because we will control with this when and what to display (app.component.html shows checking on the condition if a routing on the navbar should be displayed or not depending on a logged in/out user like 'overview', 'track time' and 'logout' should only be visible if current user is logged in => BehaviourSubject 'loggedin' equals true).
While the logOut() function only posts the message (written in the equal backend function from app.js app.post('/login', (req, res) => {}), login sends the needed data (username + token) to identify the user and sets the BehaviourSubject to 'true'. With the .navigate-method we can allocate the user to a certain component after login/logout.

The functions getToken() and getUsername() are used at every component which needs to communicate with the db like overview to fetch data or tracking to send data to identify the current user. At @Injectable({ providedIn: "root" }) we can assign the BehaviorSubject to be listened to in our root files => that's why we can use it in app.component.html

```
@Injectable({
  providedIn: 'root'
})

export class LoginService {
  loggedin$ = new BehaviorSubject(false);
  token: string = '';
  username: string = '';

  constructor(private http: HttpClient,  private router: Router) {}
  httpOptions = {···
  };

  logIn(username: string, password: string){
    this.http.post<{message: string, token: string, username: string;}>('http://localhost:3000/login', {password:
    password, username: username}, this.httpOptions)
        .subscribe((responseData) => {
          console.log(responseData);
          if(responseData.message == "Login from express.js"){

            this.loggedin$.next(true);
            this.token = responseData.token;
            this.username = responseData.username;
            this.router.navigate(['overview']);
          }
    });
  }

  logOut(){
    this.http.post<{message: string;}>('http://localhost:3000/logout', {token: this.token}, this.httpOptions)
        .subscribe((responseData) => {
          console.log(responseData.message);
          if(responseData.message == "logout successful"){
            this.loggedin$.next(false);
            this.token = '';
            this.username = '';
            this.router.navigate(['login']);
          }
    });
  }

  getToken(){
    return this.token;
  }

  getUsername(){
    return this.username;
  }
```

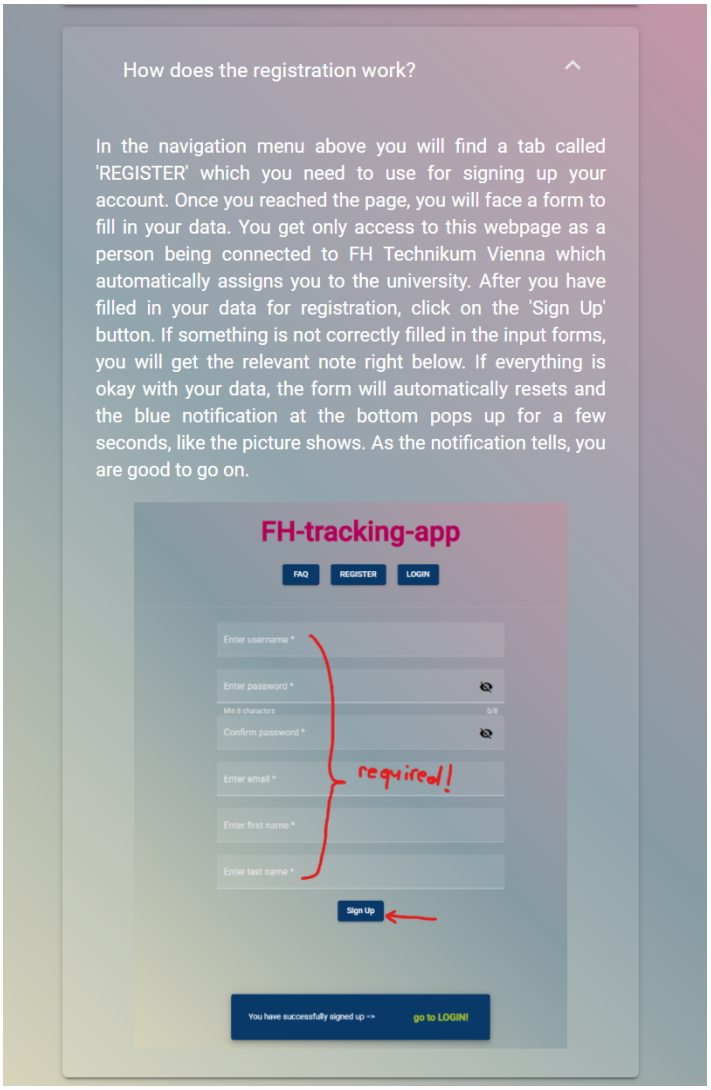
FH-TRACKING-APP

component [FAQ]:

For the FAQ no complex logic was in need, because the material component ‘mat-expansion-panel’ provides you with nearly everything that is needed. It is only necessary to define a boolean variable to use for controlling if the panels starts closed or opened as the page is visited (as you can see marked at the picture below)

The panels include text to answer the most common questions and some are supported by images/screenshots. I implemented 9 of them. The second picture displays what it looks like.

```
<mat-expansion-panel (opened)="panelOpenState = true"
                     (closed)="panelOpenState = false">
  <mat-expansion-panel-header>
    <mat-panel-title>
      How does the registration work?
    </mat-panel-title>
  </mat-expansion-panel-header>
  <p>In the navigation menu above you will find a tab called 'REGISTER' which you need to use for signing up your
  account. Once you reached the page, you will face a form to fill in your data. You get only access to this
  webpage as a person being connected to FH Technikum Vienna which automatically assigns you to the university.
  After you have filled in your data for registration, click on the 'Sign Up' button. If something is not
  correctly filled in the input forms, you will get the relevant note right below. If everything is okay with
  your data, the form will automatically resets and the blue notification at the bottom pops up for a few
  seconds, like the picture shows. As the notification tells, you are good to go on.</p>
  <div>
    <img class="faq_image" src="../../assets/img/faq_registration.png" alt="screenshot of registration">
  </div>
</mat-expansion-panel>
```



component [overview]:

Starting with the db.js we use the username as an argument to filter for data belonging to the logged in user and send it via app.js function to the frontend. While getUserProfile() can use the method .findIndex I needed to loop the second db array, because we can have more than 1 input belonging to the same user. Everytime we find data going forward in the loop, we push the data into an array of objects, assigning every information to its correct position.

```
// get overview -> all tracking data regarding chosen user
getUserProfile: function(username) {
    let indexData = this.users.findIndex(i => i.username === username);
    return {
        nameData: this.users[indexData].username,
        firstnameData: this.users[indexData].firstname,
        lastnameData: this.users[indexData].lastname,
        emailData: this.users[indexData].email,
        universityData: this.users[indexData].university
    };
},

getTrackingRecords: function(username) {
    //let indexTracking = this.trackingRecords.findIndex(i => i.username === username);
    let trackingData = [];
    let j = 0;
    for(let i = 0; i < this.trackingRecords.length; ++i) {
        if (this.trackingRecords[i].username === username) {
            trackingData.push({
                roomData: this.trackingRecords[i].room,
                dateData: this.trackingRecords[i].date,
                startTimeData: this.trackingRecords[i].begin,
                endTimeData: this.trackingRecords[i].end,
                timeInSecondsData: this.trackingRecords[i].timeInSeconds
            });
        }
    }
    return trackingData;
}
```

The overview.components.ts file uses getUserData() at every call of webpage (ngOnInit(){}) to fetch the data we receive from the backend (using the loginService methods .getToken() and .getUsername() we identify the current user.
For building the table element on the html file we provide a variable with an array filled up with the column names.
Calling the loginService method .subscribe helps us keep identifying that the current user is still logged in as we call the component again and again.

```
userData: any = [];
trackingData: any = [];
displayedColumns = ['position', 'room', 'date', 'start', 'end', 'durationSeconds'];
loggedin = false;

constructor(private http: HttpClient, private loginService: LoginService) {
  this.loginService.loggedin$.subscribe(x => this.loggedin = x);
}

ngOnInit(): void {
  this.getUserData();
}

getUserData(){
  this.http.get<{message: string, userProfileData: Object, userTrackingData: Object;}>('http://localhost:3000/
  overview?token=' + this.loginService.getToken() +'&username=' + this.loginService.getUsername(),
  {'responseType':'json'})
      .subscribe((responseData) => {
        this.userData = responseData.userProfileData;
        this.trackingData = responseData.userTrackingData;
      });
}
```

As already described in the technical steps, I had some troubles understanding that I had to use the names from the db.js functions to call/bind the fetched data to display at ‘table’. For the table I used the material component, calling the fetched data via [dataSource].

FH-TRACKING-APP

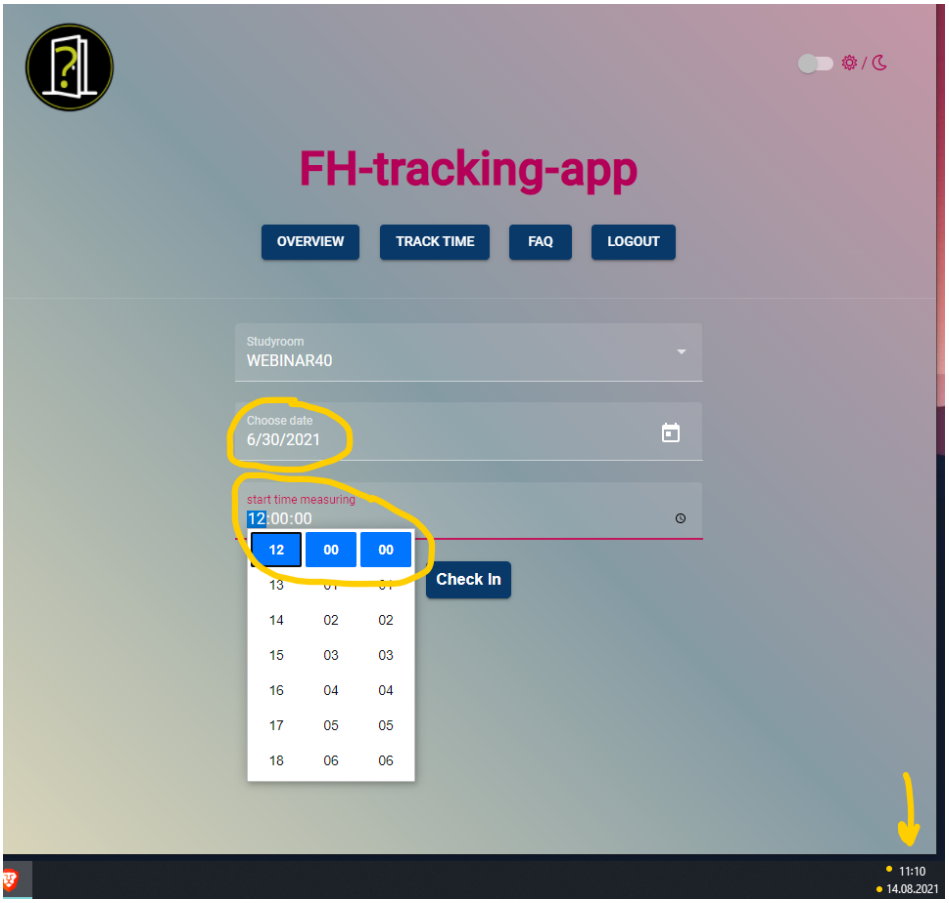
component [tracking]:

I started coding the tracking component with an intern array of rooms which lasted. I did not implement the json file, provided as demo data from you. To track time you can select a room as a dropdown list, date and time as input from ‘picker’ using the classic angular material components.

The date-picker allows to select every day in the past and the present day (so theoretically a forgotten time to track can be re-done with selecting a past day with correct time => forgot to track my time last week? let’s redo it on sunday with the settings from the desired day). By disabling a future day, no cheating can be executed (line 13 calls a variable from .ts file)

src > app > tracking > tracking.component.html > div#tracking_body.container > form#tracking_form.example-form > mat-form-field > input

```
1  <div class="container" id="tracking_body">
2      <form id="tracking_form" class="example-form" [formGroup]="trackingForm" (ngSubmit)="OnSubmit()">
3          <mat-form-field appearance="fill">
4              <mat-label>Studyroom</mat-label>
5              <mat-select formControlName="roomInput">
6                  <mat-option *ngFor="let room of rooms"  [value]="room">{{room}}</mat-option>
7              </mat-select>
8              <mat-error *ngIf="trackingForm.get('roomInput')?.errors?.required">Input required</mat-error>
9          </mat-form-field><br>
10
11         <mat-form-field appearance="fill">
12             <mat-label>Choose date</mat-label>
13             <input matInput [matDatepicker]="picker" [max]="maxDate" readonly formControlName="dateInput">
14             <mat-error *ngIf="trackingForm.get('dateInput')?.errors?.required">Input required</mat-error>
15             <mat-datepicker-toggle matSuffix [for]="picker"></mat-datepicker-toggle>
16             <mat-datepicker #picker></mat-datepicker>
17         </mat-form-field><br>
18
19         <mat-form-field class="example-full-width" appearance="fill">
20             <mat-label>start time measuring</mat-label>
21             <input matInput type="time" step="1" formControlName="timeInput" id="tracking_time_input">
22             <mat-error *ngIf="trackingForm.get('timeInput')?.errors?.required">Input required</mat-error>
23         </mat-form-field><br>
24
25         <button type="submit">Check In</button>
26     </form>
27     <button type="button" id="button_checkout" (click)="postTracking(); clearInput();">Check Out</button>
28 </div>
```

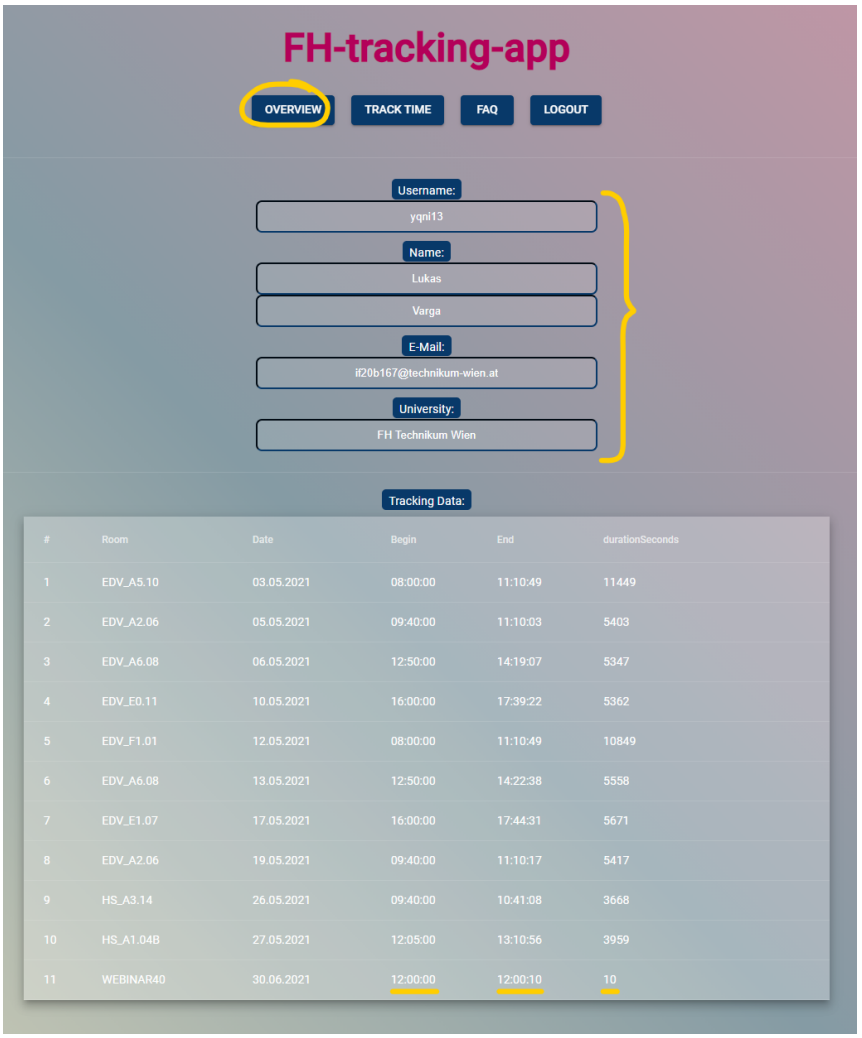


FH-TRACKING-APP

By choosing a time to track there are 2 options on how to do:
1) You choose the current time which will automatically display as first choice => check in and out and your time will be displayed as it happened for real
2) You can choose another time (as mentioned before if you need to redo it because it has been forgotten etc) and the overview will display your chosen time with the equal end time, depending on your time measured until check out.

The tracking.component.ts uses 3 functions. To divide the time tracking into 2 parts we have a function called on submit, which saves the current timestamp + needed regular expression, chosen date+time and enables/disables the two different buttons (check-in/out) and a function to 'count' time and calculate the correct end time for the checkout
(postTracking()). Basically it divides the time difference between check-in and check-out into hours, minutes and seconds, adds it to the selected starting time to get the equal end time and saves the total difference in seconds to display in the overview.
And the third function resets the input fields after check-out (clearInput()).

A new registered user will start with an empty table, only showing "N/A" as placeholder for no data in the current state which will be replaced with upcoming tracked time data. This happens, because the function getTrackingRecords() from db.js asks if tracking data exists for the current user. If no data exists, the "N/A" will be used as a placeholder. Otherwise new tracking data will be pushed in as last data element.



FH-TRACKING-APP

time table (summary)

A	B	C	D	E	F
	Varga Lukas				
		sum - hours of work:		26:39:00	lead developer
#	date	start	end	duration	content of work
1	06.08.2021	11:00:00	12:15:00	01:15:00	downloaded instructions & demo data; created timetable; updated Angular+Node, created new Angular project + GitHub repository; pushed basic node data via Git commands to repository; created GitHub Kanban Board
2	06.08.2021	16:30:00	17:15:00	00:45:00	filled GitHub Kanban Board with todo's; created base form of protocol
3	07.08.2021	14:45:00	17:00:00	02:15:00	generated components (login, register, faq, overview, tracking) and reused code from semester project for login + register + validation; generated routing
4	07.08.2021	19:00:00	20:00:00	01:00:00	extended content of component register; thought about useful functions in db.js and app.js
5	08.08.2021	16:40:00	17:30:00	00:50:00	installed password-hash and random-token modules for login; added login-requirement to enable overview and tracking only for logged in users; adapted register component;
6	10.08.2021	13:00:00	15:30:00	02:30:00	overview component; debugging
7	10.08.2021	18:40:00	21:10:00	02:30:00	updated overview + tracking component logic&html; adapting db.js + app.js; tested registration/login/overview functionality and debugged errors
8	11.08.2021	10:20:00	12:20:00	02:00:00	googled + tried different time syntax to get chosen timestamp from timepicker AND timestamp from 'Check In' to calculate time difference until 'Check Out' and calculate fitting ending time to equal timepicker
9	11.08.2021	13:20:00	15:55:00	02:35:00	finished logic (tracking will take room, date and time as input -> check-out and check-in appear and disappear as wanted; data will be saved in db and displayed on overview component)
10	11.08.2021	18:00:00	19:14:00	01:14:00	tested functionality and debugged last errors; updated backend
11	11.08.2021	20:00:00	23:15:00	03:15:00	created design for every component; added toggle to header; included choice between darkmode/lightmode
12	12.08.2021	18:00:00	20:20:00	02:20:00	filled up FAQ with content (text + screenshots); added test data into db; started writing protocol
13	13.08.2021	09:10:00	10:20:00	01:10:00	protocol 'technical steps' done; protocol 'application guidelines' part registration done
14	14.08.2021	00:10:00	01:30:00	01:20:00	wrote protocol part 'application guidelines' for every component, accept 'track time', because I want to try using the json file for creating the dropdown => if successful I need to adapt db.js, overview.component.html and so forth
15	14.08.2021	09:55:00	11:20:00	01:25:00	finished documentation
16	14.08.2021	14:25:00	14:40:00	00:15:00	zip all needed files; upload to Moodle for grading

The timetable will be saved as xlsx and pdf file.

To locate all files read the README.txt file.