Mimical

Documentation

Capstone Project: Team-08 - Mimical

Daniel Kammerzell, Tim Suchan, Maxim Torgovitski, Stoil Iliev, Ved Antigen

27.01.2023

Table of Contents

Relevant Links

GitLab App Repository:
https://gitlab.com/ciis-capstone-project/winter-2022-2023/team-08/mimical
GitLab Website Repository:
https://gitlab.com/ciis-capstone-project/winter-2022-2023/team-08/mimical-website
ReadMe:
https://gitlab.com/ciis-capstone-project/winter-2022-2023/team-08/mimical/-/blob/main/README.md

Technology Used and working methods

When developing our App, one of our main constraints was the need to have it work seamlessly on both Android and iOS platforms. After researching various options, we decided to use React Native as our framework of choice. React Native allows for the creation of cross-platform mobile applications using JavaScript and React, which was already a familiar technology for our team.

For our Website, we chose to use NextJS. The reason for this decision was that we had already gained experience with it in a preliminary course and felt comfortable using it. While there are many other frameworks available, we felt that our limited experience and the need for a quick development process made NextJS the best choice for us.

Lastly, for the backend of both our App and Website, we decided to use ExpressJS. One of the main reasons we chose ExpressJS was its extensive documentation and the large number of tutorials and guides available online. Additionally, ExpressJS is lightweight and easy to use, making it a perfect fit for our project.

We used to have daily meetings as part of our Scrum process. These meetings were essential for our team to stay organized and focused on our goals.

After each Sprint, we would meet with our product owners to showcase our progress and demonstrate the work that we had completed. This was an important opportunity for the product owners to provide feedback and ensure that the work being done aligned with their vision for the product.
In addition to progress updates, we also held retrospectives after each Sprint. This was a chance for the team to reflect on the previous Sprint and identify areas for improvement. We would discuss what went well, what didn't go well, and what we could do differently in the future. This allowed us to continuously improve our process and work more efficiently.

After the retrospective, we would then plan the goals for the next Sprint and adjust our Sprint backlog accordingly. This allowed us to ensure that we were always working on the most important tasks and making progress towards our overall goal.

Overall, the daily meetings and regular retrospectives were crucial for our team's success in using the Scrum framework. They allowed us to stay on track and make steady progress towards completing our project.

Testing and Deployment

The provision of the app is currently not possible due to the considerable development effort. Three months are unfortunately not enough to fulfill all these steps. Additionally, hosting the website would need to be done at our own cost. Our current V-Server is not compatible with hosting using NextJS.

Testing the app is also a significant effort, as a large amount of time was spent in research during the development process. The scope of the project is quite large, and there are many different features and functionality that need to be thoroughly tested before the app can be deployed. Furthermore, the iterative nature of app development means that tests will need to be continually adjusted as the project progresses, which adds additional complexity to the testing process.

Limitations

1. **Limited experience in website or app development:** Our team has limited experience in developing websites or apps, which may affect the overall quality and functionality of the project.
2. **Lack of IT affinity of the product owners:** Lack of IT product owners results in minimal technical feedback and inability to conduct code reviews. In some cases, this has led to situations where clean code was not always observed or even best practices were not followed.
3. **High trial and error:** A lot of time has been spent in trial and error, which can slow down the development process and increase the risk of errors.
4. **Self-taught:** A lot of the team members had to teach themselves the necessary skills and technologies, which may lead to a lack of expertise in certain areas.
5. **Server not available:** The server was not available and had to be organized and initialized by ourselves, which added extra time and effort to the project.
6. **Large project scope:** The project has a very large scope, which leads to specialization and a lack of good code reviews.

7. **Lack of time for testing:** Due to the large scope of the project, there was not enough time to set up a proper pipeline or other testing methods, which can lead to issues with quality control.

   It is important to keep these limitations in mind when assessing the overall quality and functionality of the project. Despite these limitations, we tried to deliver a functional and high-quality product.

Changelog

App

- How to start the app:
    - Install the Expo Go app
    - Navigate to directory in terminal
    - Enter command: npm start (**if the app does not start, you have to enter npm install --legacy-peer-deps first)**
    - Follow instructions in terminal

- 2023-01-12 -- 1.0.0
    - added app icon
    - filter is fully functional
    - added sounds
    - added scenarios
    - bug fixes
    - continue is in the menu instead of homescreen
    - login is fully functional
    - register is fully functional

- 2023-01-13 -- 0.0.4
    - added modal for transitioning between tasks and the menu
    - added specific sounds and narrator to each task
    - added functionality to take pictures which get saved in the users cache
    - added automatic key generation when a patient registers for the first time

    - added additional (required) inputs for registration: names, birth date, gender
    - new registration and login design
    - made menu progress bars functional
    - made exercise completion display functional
    - added start next level button to home screen
    - task redesign
    - additional options in settings

- 2022-12-16 -- 0.0.3

    - settings are stored locally
    - possibility to change language (currently only german and english)
    - possibility to change font size (for better readability)
    - added Sound to all levels
    - implemented filter on tags in the menu
    - configured CI/CD
    - fixed bug on timer buttons
    - add all long scenarios
    - scenarios are now displayed through the contentManager instead of
    - finished adjustments for the tasks and task timer
    - level completion states are now stored in async storage and saved even after restarting the app
    - patients are able to login and register
    - patient passwords are hashed and saved in the database

- 2022-12-02 -- 0.0.2

    - added a system to store the level contents and a state that refers to the current level globally. This way the user can navigate through levels and start levels from any given point
    - fixed the bug that the camera would just return a black screen in ios devices
    - added a timer that can be started from the task component and times the users exercise. The timer appears in the camera preview and is highlighted shortly before the user has to start exercising again while he is on pause.
    - the date of the last user login is saved, which also updates whenever a user logs in again
    - user can look at progress
    - menu (unlocked/completed exercises are marked)

    - settings page (functionality)
    - light/dark mode

- 2022-11-18 -- 0.0.1

    - app created with react native, specifically expo
    - start page (containing: continue practice, scenario overview, log in, cam preview, settings, notifications, calendar)
    - menu (UI design, navigation between pages)
    - settings page (layout)
    - notifications: option to receive notifications as reminders for the exercises; can be set for the morning (11 AM), evening (18 PM), or both; button to deactivate all notifications;
    - calendar: option to create a reminder in device's calendar app, by importing an event in the calendar submenu
    - front cam preview for exercises

- Repository for the app: https://gitlab.com/ciis-capstone-project/winter-2022-2023/team-08/mimical

Website

- How to start the website:

    - Once in the repository, install the necessary dependencies using the "npm install" command
    - After the dependencies are installed use "npm run dev" to start the application
    - Add the ".process.env" to the repository (https://drive.google.com/file/d/11x0xJ-RxX5Fm3aoCA5qGasrH-VZsZQhJ/view?usp=share_link)
    - It is important to make sure that the file is named ".process.env", because the file will be renamed to "process.env" after the download!

- 2023-01-20 -- 1.0.0

    - the repository for the website can be found here: https://gitlab.com/ciis-capstone-project/winter-2022-2023/team-08/mimical-website

    - the website design has now been adapted to the design of the app
    - ChartJS was implemented and fully functional on the last 365 days seen
    - comment function was introduced (adding and deleting comments)

- 2023-01-13 -- 0.0.4

    - the repository for the website can be found here: https://gitlab.com/ciis-capstone-project/winter-2022-2023/team-08/mimical-website
    - therapists are able to add patients if a key is provided by the patient
    - therapists can edit data when adding a patient
    - therapists can edit data after they added a patient
    - charts are now implemented on the website
    - design changes

- 2022-12-16 -- 0.0.3

    - the repository for the website can be found here: https://gitlab.com/ciis-capstone-project/winter-2022-2023/team-08/mimical-website
    - therapists are able to login
    - middleware for protected routing is implemented
    - added database to the project
    - patients are displayed exclusively to the associated therapist
    - patient add dummy
    - new express fetches and calls

- 2022-12-02 -- 0.0.2

    - the repository for the website can be found here: https://gitlab.com/ciis-capstone-project/winter-2022-2023/team-08/mimical-website
    - new design
    - created patient creation page
    - therapist is able to create new patients
    - created dynamic patient cards
    - created mysql database
    - added connection to mysql database (right now still local)
    - added express as backend
    - added new login page

- 2022-11-18 -- 0.0.1

    - created project with NextJS
    - the repository for the website can be found here: https://gitlab.com/ciis-capstone-project/winter-2022-2023/team-08/mimical-website
    - created login page with possibility to save entered values
    - created patients page with a search bar for entries which are contained inside a JSON file
    - created add patients page (which can be entered through the corresponding button located in the patients page) with the possibility to save entered values for future backend
    - created navbar with desktop and mobile view for responsiveness
    - created footer with links to facebook, instagram and linkedin
    - added a temporary CSS Layout
    - created homepage and settings page (empty)

Database

- How to add the database to the project:

    - Currently there are two backends (both expressJS) for testing purposes.
    - On the WEBSITE, the backend is started automatically when "npm run dev" is executed.
    - On the APP, on the other hand, it is important that two processes are started. Once for the app ("npm start") and once for the server ("npm run backend").

1. Add the ".process.env" to the repository (https://drive.google.com/file/d/11x0xJ-RxX5Fm3aoCA5qGasrH-VZsZQhJ/view?usp=share_link)
2. Launching the app/website in the normal way

- 2023-01-20 -- 1.0.0

    - the database now stores all the progress of users

- 2023-01-13 -- 0.0.4

- 2022-12-16 -- 0.0.3
    - added the design of the database
    - added tables to the database
    - added patients and therapists to the database

Software - Documentation

App

Back-end

The back-end we chose to operate on for the app is **ExpressJS**, since our front-end is based on JavaScript. As we need to build the entire project from scratch and are short on time, it makes little sense to look at back-end solutions that are based on something else other than JavaScript. Furthermore, there is a lot of documentation for Express JS, which saves a lot of effort. In an earlier version, a **PHP** server was set up to connect the client with the database. This did not offer enough flexibility and worked rather poorly with JavaScript, which is why it was decided the back-end connection would be done with an *Express* server, using *Node JS*.

Taking a look at the ***'/backend'*** folder, there are two JavaScript files: '***db.js***' and '***server.js***'.

- '***db.js***' contains a connection to the database that passes on the login data for our server. Here ***mysql*** and ***dotenv*** are imported. The first one is needed since this is the database we are using. The second one is reading the database login info, which is saved on a separate file named '***.process.env***', located in the root directory of the project.
- '***server.js***' is relatively longer. Firstly, the required imports are declared. These include the **Express** application itself, ***crypt*** and ***db.js***, among other, all listed here:

```
1   const express = require("express");
2   const bodyParser = require("body-parser");
3   const mysql = require("mysql");
4   const dotenv = require("dotenv");
5   const cors = require("cors");
6   const bcrypt = require("bcrypt");
7   const db = require("./db");
```

bodyParser is used to handle incoming JSON requests, cors for handling cross-origin resource sharing and ***bcrypt*** for hashing passwords.

The server is listening on **port 3000** and whenever a request is made it routes the request to Next JS.

Firstly, in '***/api/signin***', we check if the email is present in '**patients'** from the database. If so, we compare the provided password with the saved hashed password, using ***bcrypt***. If these match,

the user can log in. An additional MySQL query is sending the patient's ID to the client:

```
64                  if (response) {
65                    //Password is correct
66                    console.log("Angemeldet");
67                    //Fetch and send the patient ID
68                    let sqlid = "SELECT ID FROM patients WHERE email = ?";
69                    let queryid = db.query(
70                      sqlid,
71                      [data.Email],
72                      (err, results) => {
73                        if (err) throw err;
74                        if (results) {
75                          res.send(results);
```

In '***/api/key***' we can retrieve the patient's key that was generated upon registration from '**patients'** in the database. Again first test if the provided email is saved in the database, then compare passwords. Only then send the key to be displayed to the user:

```
117                 if (response) {
118                   //Password is correct
119                   //Fetch and send the patient ID
120                   let sqlkey =
121                     "SELECT therapistAddKey FROM patients WHERE email = ?";
122                   let queryid = db.query(
123                     sqlkey,
124                     [data.Email],
125                     (err, results) => {
126                       if (err) throw err;
127                       if (results) {
128                         res.send(results);
```

'***/api/signup***' handles the signing up requests. The data is saved on '**patients'** in the database. We need to test if the email is already in use, after which whether the patient has been assigned to a therapist. Then the email, names, gender, birth date, password, key and key expiration date are saved to the database. The password is hashed (salted 10 times) with ***bcrypt***:

```
153     //Hash password
154     hash = await bcrypt.hash(data.Password, 10);
```

The function ***makeKey*** is used to generate a string key of variable length, in our case we use 14 characters:

```
156   //Generate key
157   function makekey(length) {
158     var result = "";
159     var characters =
160       "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
161     var charactersLength = characters.length;
162     for (var i = 0; i < length; i++) {
163       result += characters.charAt(Math.floor(Math.random() * charactersLength));
164     }
165     return result;
166   }
```

This ***Date*** object is then converted to a *string* and reformatted into the 'YYYY-MM-DD' format, before saving it in the database.

'***/api/progress***' is used to save the current progress of the patients in our 'patient-progress' table of the database. We count when an exercise is targeting the upper or lower face with two temporary variables – ***up*** and ***low***. The same procedure as in ***signup*** is used to save the date of completion of every exercise – converting to *string* and reformatting. We test whether the patient ID is already present on '**patient-progress**' in the database.

If so, we then update the count of completed upper/ lower tasks and save the date of the completion in the '***dateX***'-th column of the patient's row, where **X** is the numerical representation of a specific exercise.

```
285         let uploadData =
286           "UPDATE `patient-progress` SET `date" +
287           Progress +
288           "`= '" +
289           progressDate +
290           "', `upperCount`= `upperCount` + '" +
291           up +
292           "', `LowerCount`= `LowerCount` + '" +
293           low +
294           "'" +
295           " WHERE `patient-progress`.`patientID` = " +
296           data.PatientID;
```

If not, we need to do the same procedure, only insert a whole new patient row in '**patient-progress**', not just update it, including possible count of just completed exercises.

```
322         let insertID =
323           "INSERT INTO `patient-progress` (`patientID`, `date" +
324           data.ContentProgress +
325           "`, `upperCount`, `lowerCount`) VALUES ('" +
326           data.PatientID +
327           "', '" +
328           progressDate +
329           "', '" +
330           JSON.stringify(up) +
331           "', '" +
332           JSON.stringify(low) +
333           "')";
```

To follow progress, we use several functions in '**alternativeTask**' (see Progress and completion storage). First the patient's ID is retrieved from async storage (it was saved when logging in, see '***signin.js***') and set it as the ***PatientID***.

```
137   //Get Patient ID from async storage
138   const getPatientID = async () => {
139     try {
140       const PatientID = await AsyncStorage.getItem("ID");
141       if (PatientID !== null) {
142         setPatientID(PatientID);
143         //console.log(PatientID);
144       }
145     } catch (error) {
146       console.log("Can't retrieve data from async storage");
147     }
148     //console.log("Done.");
149   };
```

With an Axios Post request we then perform the MySQL queries as mentioned above in '***/api/progress***', whenever a patient clicks the '***Zurück'*** or '***Weiter'*** after completing an exercise.

```
124     await axios({
125       method: "post",
126       data: {
127         ContentProgress: ContentProgress,
128         PatientID: PatientID,
129       },
130       // Must be changed depending on device for testing
131       url: "http://192.168.1.98:3000/api/progress",
132     })
133       .then((res) => console.log(res))
134       .catch((err) => console.log(err));
135   };
```

Registering a new user

signup.js:

This file is used for the registration of new users. A user is required to input their **first** and **last names**, **email**, **gender**, **birthdate** and choose a **password**.

It starts by importing the necessary modules and libraries.

```
// Import react native
import {
  Button,
  Pressable,
  ScrollView,
  Switch,
  Text,
  TextInput,
  useColorScheme,
  View,
} from "react-native";
import React, { useEffect, useState } from "react";
import {
  light_primary_color,
  dark_primary_color,
  light_background_color,
  dark_background_color,
  green,
  gray5,
  dark_gray5,
} from "../components/styles.js";
import axios from "axios";

// import components
import styles from "../components/styles.js";
```

After that the necessary state variables are declared, followed by some additional state variables, required for the selection (also see ***Gender*** below). Some styling is applied as well.

```
// signup state variables
const [Prename, setPrename] = useState("");
const [Name, setName] = useState("");
const [Email, setEmail] = useState("");
const [Gender, setGender] = useState("");
const [Birthdate, setBirthdate] = useState("");
const [Password, setPassword] = useState("");
const [ConfrirmPassword, setConfirmPassword] = useState("");

// selection state variables
const [optionIsEnabled1, setOptionIsEnabled1] = useState(false);
const [optionIsEnabled2, setOptionIsEnabled2] = useState(false);
const [optionIsEnabled3, setOptionIsEnabled3] = useState(false);
```

There are no limitations on first and last **names**, except that these fields must not be empty.

```
121     } else {
122       //Name input fields should not be empty
123       if (
124         Prename == 0 ||
125         Prename.length == 0 ||
126         Name == 0 ||
127         Name.length == 0
128       ) {
129         alert("Vollständige Namen fehlen");
```

The **email** input must not be empty. Additionally, a regular expression checks whether the provided text is a valid email:

```
 97     // Regular expression to validadate email
 98     var checkEmail = RegExp(
 99       /^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,3})$/i
100     );
```

A **password** must consist of at least 8 characters, at least one capital letter, at least one number and at least one special character (/*:.!,@# etc.) An empty space is also not allowed. Several *if* statements and a regular expression test, whether these requirements are met:

```
114     // Password validations
115     else if (Password.length < 8) {
116       alert("Mindestens 8 Zeichen");
117     } else if (!/[ `!@#$%^&*()_+\-=\[\]{};':"\\|,.<>\/?~]/.test(Password)) {
118       alert("Mindestens ein Sonderzeichen");
119     } else if (/[ ]/.test(Password)) {
120       alert("Ohne Leerschritt");
121     } else {
```

To ensure that the user has not mistyped his password, a second (confirmation) password prompt validates that the two passwords match.

The names, email and password fields are all using the standard React Native ***TextInput*** (however the password and password confirmation fields have ***secureTextEntry*** enabled), so that whenever the text is changed, ***setPrename***, ***setName***, ***setEmail***, ***setPassword*** and ***setConfirmPassword*** also change the corresponding variables ***Prename***, ***Name***, ***Email***, ***Password*** and ***ConfirmPassword***.

```
201               placeholder="Vorname"
202               onChangeText={(Prename) => setPrename(Prename)}
```

The ***Gender*** choice, consisting of three options: ‘male’, ‘female’ and ‘diverse’, is a ***Selection*** object, similar to the ones created by *Selection.js* in */components*, but with these three options, instead of two. Here another React Native component *Dropdown-Picker* was also considered, but it proved too inflexible in terms of styling that would clash with the rest of the visuals of the app. When the chosen button is clicked, ***setGender*** changes the value of Gender:

```
64    const selectOption2 = () => [
65      setOptionIsEnabled1(false),
66      setOptionIsEnabled2(true),
67      setGender("w"),
68      setOptionIsEnabled3(false),
69    ];
```

The ***Birthdate*** input was originally a React Native ***Date-Time-Picker-Modal*** element, however it seemed to not work well on iOS and was swapped for a simpler ***TextInput***, combined with a regular expression to validate that the text is a date. The 'YYYY-MM-DD' format was chosen as the easiest to directly transfer to the database. The date is transferred to Birthdate via ***setBirthdate*** when the text changes.

```
102      // Regular expression to validadate birthdate format
103      var checkBirthday = RegExp(
104        /\d{4}-(?:0?[1-9]|1[012])-(?:0?[1-9]|[12][0-9]|3[01])*/
105      );
```

Every input is wrapped in ***ScrollView***, so that no element becomes hidden or inaccessible for the user. Additional styling, in line with the general visual style of the app, is applied to every input and button.

After all the input test have passed an **Axios** Post request is sent to the server to transfer the data into the database:

```
140              //Send Axios Post request
141              await axios({
142                method: "post",
143                data: {
144                  Email: Email,
145                  Password: Password,
146                  Name: Name,
147                  Prename: Prename,
148                  Birthdate: Birthdate,
149                  Gender: Gender,
150                },
151                // Must be changed depending on device for testing
152                url: "http://192.168.1.98:3000/api/signup",
153              })
154                .then((res) => console.log(res))
155                .catch((err) => console.log(err));
```

After the user has filled in all the necessary data and clicked on 'Registrieren', this asynchronous request takes all the saved user data and sends it to the server. In this case the server is running on a local network.

If everything is successful, the app navigates to the *Login* screen (see ***signin.js***). A **key** is automatically generated (see ***server.js***).

Please note: When deploying this app (or testing it), it will be necessary to change the address to the remote address, where the server would be running.

Logging In as a user

signin.js:

This file is used for the login of already registered users. A user is required to input their **email** and **password**.

It starts by importing the necessary modules and libraries:

```
import {
  Button,
  Pressable,
  ScrollView,
  Switch,
  Text,
  TextInput,
  useColorScheme,
  View,
} from "react-native";
import React, { useEffect, useState } from "react";
import {
  light_primary_color,
  dark_primary_color,
  light_background_color,
  dark_background_color,
  green,
  gray5,
  dark_gray5,
} from "../components/styles.js";
// import components
import styles from "../components/styles.js";
import axios from "axios";
import AsyncStorage from "@react-native-async-storage/async-storage";
```

Some styling is imported, after which several state variables are declared. The inputted data is saved here.

```
  //signin state variables
  const [Email, setEmail] = useState("");
  const [Password, setPassword] = useState("");
  const [PatientID, setPatientID] = useState("");
```

The **email** and **password** must not be empty. If that is the case, inside ***submit*** an **Axios** Post request is sent to the server to transfer the data into the database:

```
48      //Send Axios Post request
49      await axios({
50        method: "post",
51        data: {
52          Email: Email,
53          Password: Password,
54        },
55        // Must be changed depending on device for testing
56        url: "http://192.168.1.98:3000/api/signin",
57      })
58        .then((res) => {
59          //Save patient ID on device storage
60          setPatientID(res.data[0].ID);
61          console.log(res.data[0].ID);
62          //Navigate to next screen if authentications are valid
63          navigation.navigate("Menu");
64        })
```

Please note: When deploying this app (or testing it), it will be necessary to change the address to the remote address, where the server would be running.

If the request was successful, ***PatientID*** is also set to the fetched value (see ***savePatientID*** below). After that the app navigates to the main screen of the app.

Retrieving key:

In order to be assigned to a therapist, each patient has to provide a 14-character key to a therapist. A key is generated automatically after a successful registration.

As with logging in, **email** and **password** must not be blank. If that is the case, inside ***getKey*** an **Axios** Post request is sent to the server to retrieve the data from the database:

```
81      await axios({
82        //Send Axios Post request
83        method: "post",
84        data: {
85          Email: Email,
86          Password: Password,
87        },
88        // Must be changed depending on device for testing
89        url: "http://192.168.1.98:3000/api/key",
90      })
91        .then((res) => {
92          //Show patient key
93          alert("Ihr Key ist: " + res.data[0].therapistAddKey);
```

This key is then shown to the patient in a separate alert window.

In ***savePatientID*** we save the patient's ID, which we received when logging in, to device storage:

```
106   const savePatientID = async (PatientID) => {
107     try {
108       await AsyncStorage.setItem("ID", JSON.stringify(PatientID));
109       console.log("Saved Patient ID" + " : " + PatientID);
110     } catch (error) {
111       console.log("Can't save data to async storage");
112     }
113   };
```

A ***UseEffect*** is also used, so that the ID is correctly saved immediately after the ID is retrieved from the database.

The email and password inputs are all using the standard React Native ***TextInput*** (however the password field has ***secureTextEntry*** enabled), so that whenever the text is changed, ***setEmail*** and ***setPassword*** also change the corresponding variables ***Email*** and ***Password***. There are two ***Pressable*** buttons – '**Anmelden**' and '**Key Anzeigen**', which use ***onChangeText*** to call the corresponding functions:

```
152           placeholder="Passwort"
153           onChangeText={setPassword}
154           secureTextEntry={true}
```

The Filter

In the Menu page the scenarios are always rendered from the *filteredKeyArray*, *filteredKeyArray* is a useState hook which means the menu page will be rerendered if it changes. It includes all keys from *defaultScenarios* in *contentManager.js* that pass the current filter.
All keys from *defaultScenarios* in *contentManager.js* are also stored in *keyArray* regardless of any filters.
The current filter activations are stored in the *tagStates* object:

```
const tagStates = {
 'Obere Gesichtshälfte': false,
 'Untere Gesichtshälfte': false,
 'Langes Szenario': false,
 'Kurzes Szenario': false
}
```

The objects keys refer to the filters condition and their entries to the current state. When the page is loaded no filter is active ergo all entries are set to false.
Filter buttons are rendered from the keys of this object. On press they set *filteredKeyArray* to all entries of *keyArray* that return true for *checkTags(scenarioKey).*

```
const checkTags = (scenarioKey) => {
  trueTags = Object.keys(tagStates).filter((tagState) =>
```

```
  tagStates[tagState]);
  return trueTags.every(tag =>
  getTags(scenarioKey).includes(tag));
}
```

checkTags(ScenarioKey) receives a scenarioKey and returns true if all filter conditions that are true in *tagStates* are included in the “tags” array of the scenario the *scenarioKey* refers to.
This also implies that no filtering happens if no filter is set to active by the user because there is no condition that has to be passed for a *scenarioKey* to be included in *filteredKeyArray.*

The Content Manager file

The *contentManager.js* is a development created to store level data while also keeping the way this data is stored as flexible as possible.
The Data is stored on three levels. First, there are content objects of the following structure:

```
const exampleContent = {
  "baseText": "part of the story a user is reading during a
                level that does not yet refer to its task",

  "highlightedText": "part of the story a user is reading
                 that refers to their task",

  "task": "the task as displayed on the task screen in
              imperative form",

  "sound": require("path to contents audio file")
}
```

Those contents are then stored in the *allContents* object where they receive integers as indices.

Scenarios are acquired from the *defaultScenarios* object. It’s structure is:

```
const defaultScenarios = {
  "ScenarioKey": {
     "tags": [<The Scenarios tags>],
```

```
    "indices": [<the indices of the scenarios contents in
                order>],
    "icon" : <The Scenarios Icon>
},
```

The ScenarioKey is the name of the scenario e.g. “Umzug”. Using this tag all of the relevant information of a scenario can be accessed.

Additionally it stores the three global state variables, all those variables refer to the state the user is in while playing the game:

```
let currentSequence = [<Indices of the contents in the sequence the user is currently
playing in order>];

let currentContent = <Index that reflects the user's current position in currentSequence
stored as Int>;

let currentScenario = <Key of the Scenario the user is currently playing>
```

- the reason the current scenario is stored somewhat redundantly is that it was initially planned to include custom scenarios where the scenarioKey might not have made sense.
- global states are used even though this is usually considered bad practice. However the game state has to be accessed from a variety of pages.
- A possibly better way of doing this would be to pass the same info between pages using react navigation params. However, at the time we could not get react navigation params to work properly.
- Pages like *level.js* then pull their data using functions such as get current text which returns the baseText that *currentContent* refers to.

In order to keep a reasonable degree of control the state variables are only accessible through functions exported by the contentManager.js file
The ContentManager exports a number of functions used to access state variables as well as contents. This approach also has the benefit of being able to change the way content is stored later in the development process without touching the rest of the code.
The Content manager provides important functions for modifying states or accessing content such as *startLevel(start, scenarioKey)* Function which is used to start a level/ go to the next level.

Progress and completion storage

We wanted the user to be able to see the progress of the scenarios and pick up where they left off when they used the app before. We couldn't just store this information in normal variables as they would be reset on an app restart.
For this use case react native offers an api called async storage it offers a way to store key value pairs persistently over the device. The stored entry has to be a string. This means other data structures have to be stringified and parsed.

The completed levels are stored in an object at the "@completions" key the object itself stores the content id's of all completed contents as keys and along with a "completed" string as their entries. The id's of contents that have not been completed do not appear in this object at all.
It would have been sufficient to store completed content's id's as an array but async storage does not provide a simple solution to modify arrays.
All completion related entries are stored to async storage in the *alternativeTask.js* file.
The workflow in *alternativeTask.js* is the following:

- the *"@completions"* object is fetched from async storage. Its content is stored to the *completions* use state hook (this is done to later avoid redundant completion savings)

```
102  const fetchCompletions = async () => {
103    try {
104      const item = await AsyncStorage.getItem("@completions");
105      if (item) {
106        setCompletions(JSON.parse(item));
107      }
108    } catch { }
109  }
110
```

- this is triggered on a render of *alternativeTask* using *useEffect*

```
113  useEffect(() => {
114    fetchCompletions();
115  }, []);
116
```

- When a task has been completed (see how this is determined in the chapter "timer") the ID of the current content (see "content manager") is stored to "last task" in async storage

```
231      saveAsLast(getCurrentSequence()[getCurrentContent()]);
```

```
152  const saveAsLast = async (lastContent) => {
153    try {
154      await AsyncStorage.setItem("lastTask", lastContent.toString());
155      console.log("saved completed succesfull" + " : " + "last" + lastContent);
156    } catch (error) {
157      console.log("cant save data to async storage");
158    }
159  };
```

- If the current content is not stored as completed yet an object of the form {contentID : “completed”} is merged to the object at *“@completions”* in async storage.

```
225      if (!Object.keys(completions).includes(getCurrentSequence()[getCurrentContent()])) {
226        saveAsCompleted(ifCompleted);
227      }
```

```
52    const ifCompleted = { [thisContent]: "completed" };
```

```
90  const saveAsCompleted = async (completedContent) => {
91    try {
92      await AsyncStorage.mergeItem("@completions", JSON.stringify(completedContent));
93      console.log("saved completed succesfull");
94    } catch (error) {
95      console.log("cant save data to async storage");
96    }
97  };
```

The completion related entries are fetched and processed at *menu.js*. The workflow for this is:

- the *@completions* object is fetched from async storage and saved to the *completionStates* useState hook.

```
93   const fetchCompletionStates = async () => {
94     try {
95       const item = await AsyncStorage.getItem("@completions");
96       if (item) {
97         setCompletionStates(JSON.parse(item));
98       }
99     } catch { }
100  }
```

- When data is fetched *setCompletionStates* is adjusted triggering a rerender of the menu page because *setCompletionStates* is a use Effect hook
- This fetch and rerender process is either triggered on render using *useEffect*

```
137  useEffect(() => {
138    fetchCompletionStates();
139    fetchLastTask();
140    getData();
141  }, []);
```

- or when the page is in focus of react-navigation using *useFocusEffect*

```
83  useFocusEffect(
84    useCallback(() => {
85      fetchCompletionStates()
86      fetchLastTask();
87      getData();
88    }, [])
89  );
```

- in terms of progress related async storage entries menu fetches lastTask and completions because there are two progress related functions in *menu.js*
- First one is the progress bar and progress display in the exercise buttons
- for each Scenario accessed by its key an array of the completion states of its content is computed

```
105   const getCompletionsByScenario = (scenarioKey) => {
106     const scenario = getScenario(scenarioKey);
107     return scenario.filter(isCompleted);
108   }
```

```
112  const isCompleted = (index) => {
113    try {
114      return completionStates[index.toString()] == 'completed';
115    }
116    catch {
117      return false;
118    }
119  }
```

- The try catch in *isCompleted()* is not necessary and is only there because this function initially directly fetched from async storage
- The resulting array of *getCompletionsbyScenario()* is passed to the corresponding scenario

```
205                 completions={getCompletionsByScenario(scenarioKey)}
```

- In *scenario.js* two things happen with the completions. Firstly the progress bars values are set in accordance with the completion object using its props.

```
71      <ProgressBar exercises={scenarioLength} progress={props.completions.length} />
```

- secondly the completion states for the exercises it contains are taken from the *props.completions* array and passed down to the exercises as props

```
82              completed={props.completions.includes(getScenario(name)[iterate])}
```

- Exercises are then rendered accordingly
- The Second progress-related function in *menu.js* is the continue function, facilitated by using the *“lastTask”* entry.
- Last Task is fetched, the entry incremented and saved to nextTask and the label *fetchCompleted* is set to true. The label is used to decide whether the fetch was successful, which isn't the case when no level has been played yet.

```
124  const fetchLastTask = async () => {
125    try {
126      const item = await AsyncStorage.getItem('lastTask');
127      if (item) {
128        setNextTask(parseInt(item) + 1);
129        setFetchCompleted(true);
130        console.log("FETCH COMPLETED")
131      }
132    } catch { }
133  };
```

- this fetch is triggered on menu render and focus as explained above for the completions fetch
- Using the index of the next Task the user has to complete an exercise is rendered on top of the menu screen if *fetchCompleted* is true

```
182          {fetchCompleted && (
183            <Exercise
184              level={getScenario(getScenarioFromTask(nextTask)).indexOf(nextTask)}
185              key={nextTask}
186              icon={getIcon(getScenarioFromTask(nextTask))}
187              navigation={navigation}
188              unlocked={true}
189              completed={isCompleted(nextTask)}
190              scenarioKey={getScenarioFromTask(nextTask)}
191              fromHomeScreen={true}
192            ></Exercise>
193          )}
```

- using the *fromHomeScreen* prop the exercise is rendered differently than the other exercises on menu

Timer

The timer is displayed on the camera while the user performs a task. It instructs the user to train/pause for specific amounts of time.

It always includes three train and pause periods. That means the user trains, pauses, trains, pauses, trains to complete a task. (in lifting wording: Three sets of 10 seconds with 10 seconds relaxation)

```
19  let trainDuration = 10;
20  let pauseDuration = 10;
```

- The train and pause durations in seconds are defined at the beginning of the *alternativeTask.js* file.
- the *currentTime* (current Second) of the timer is stored as a useState hook. That way if it changes the page is rerendered and the change is displayed.

```
40  const [currentTime, setCurrentTime] = useState(trainDuration);
```

- There is a number of additional useState hooks that refer to various states the timer can be in
- !IMPORTANT DISTINCTION! pause refers to the state in which a user pressed the pause button and the timer stops running. Relax refers to the state where the user relaxes his muscles waiting for the next training period. In this state the timer is running. This also means that *pauseDuration* should be renamed to “relaxDuration”. With this in mind, these are the state hooks the timer uses.

```
41  const [taskRunning, setTaskRunning] = useState(false);//-> decides wether the timer should run at a given moment
42  const [onPause, setOnPause] = useState(false);//-> refers to the state after the user paused the training using the pause button
43  const [relaxState, setRelaxState] = useState(false); //-> refers to the delibarite relaxation pauses between training sets
44  const [informState, setInformState] = useState(false); //-> last three seconds of relax where the timer is displayed bigger as a reminder
45  const [repCounter, setRepCounter] = useState(0);//-> counts how often the training has been completed
46  const repititions = 3;//-> number of repititions the user has to perform to complete the task
```

- If the user presses play on the task screen the *play()* function is triggered:

```
65  const play = () => {
66    if (!onPause) {
67      setCurrentTime(trainDuration);
68      setTaskRunning(true);
69      setOnPause(false);
70      setInformState(false);
71    } else {
72      setTaskRunning(true);
73      setOnPause(false);
74    }
75  };
```

- This either starts the game or resumes it if *onPause* is true
- accordingly there the *pause()* function is triggered when a user presses the pause button

```
177   const pause = () => {
178     setTaskRunning(false);
179     setOnPause(true);
180   };
```

- The functionality happens in a useEffect where *setInterval* reduces *currentTime* by one every second. Additionally some breakpoints are implemented in this *useEffect* where the timer switches states.

```
196 useEffect(() => {
197   let interval = null;
198   if (taskRunning) {
199     interval = setInterval(() => {
200       removeIncrementReplace();
201     }, 1000);
202   } else if (!taskRunning) {
203     clearInterval(interval);
204   }
205   //the system goes into relax state
206   if (currentTime == 0 && taskRunning && !relaxState) {
207     setCurrentTime(pauseDuration);
208     setRelaxState(true);
209   }
210   //the system goes back to train state/ stops the task if the counter has reached 3
211   if (currentTime == 0 && taskRunning && relaxState) {
212     setRelaxState(false);
213     if (repCounter == repititions - 1) {
214       setRepCounter(0);
215       setTaskRunning(false);
216       setInformState(false);
217       clearInterval(interval);
218     } else {
219       setRepCounter((repCounter) => repCounter + 1);
220       play();
221     }
222   }
223   if (currentTime == 0 && repCounter == repititions - 1) {
224     setTaskRunning(false);
225     setCurrentTime(0);
226     //making sure not to double store completed levels in async storage
227     if (!Object.keys(completions).includes(getCurrentSequence()[getCurrentContent()])) {
228       saveAsCompleted(ifCompleted);
229     }
230
```

```
233     saveAsLast(getCurrentSequence()[getCurrentContent()]);
234     // nextLevelFunction();
235     setModalVisible(true);
236   }
237   if (currentTime == 3 && taskRunning && relaxState) {
238     setInformState(true);
239   }
240   return () => clearInterval(interval);
241 }, [taskRunning, currentTime, relaxState, informState]);
```

- The interval runs continuously if statements do case distinctions for any state changes as described in their comments. Line 223 performs some additional actions after a task is completed.

- The reason that the timer has to run inside a useEffect hook is, that the timer is performed as a side effect and the page has to be rerendered without user interaction.
- Something worth explaining here is the *removeIncrementReplace()* function in line 200. This function removes *currentTime* from the screen, increments it by one and places it in the same location again. This is done so that a layout animation is triggered.

```
165   const removeIncrementReplace = () => {
166     setRemoved(true);
167     LayoutAnimation.configureNext({
168       duration: 300,
169       create: { type: "spring", property: "scaleY", springDamping: 0.8 },
170     });
171     setCurrentTime((currentTime) => currentTime - 1);
172     setRemoved(false);
173   };
```

Camera

A camera component is provided by an expo api. However there are two things required for it to work properly. First the user's camera usage permission has to be enquired. This is done using the *requestCameraPermissionsAsync* function provided by expo camera.

```
86  useEffect(() => {
87    (async () => {
88      const status = await Camera.requestCameraPermissionsAsync();
89      setHasPermission(status["granted"]);
90      try{
91      const mediaLibraryPermission = await MediaLibrary.requestPermissionsAsync();
92      }
93      catch{}
94      setHasMediaLibraryPermission(status["granted"]);
95    })();
96  }, []);
```

Additionally, as android devices might support different aspect ratios in their front cameras we wrote a function that finds the devices supported ratio that's closest to the goal of a 4:3 aspect ratio and uses this.

```
150   const findRatio = async () => {
151     if (Platform.OS === 'android') {
152       const ratios = await Camera.getSupportedRatiosAsync();
153       if (('4:3') in ratios) {
154         setRatio('4:3');
155         setDecimalRatio(1.33333);
156       }
157       else {
158         let min = 5;
159         let bestRatio;
160         for (let ratio of ratios) {
161           const splitted = ratio.split(':');
162           const decimalRatio = parseInt(splitted[0]) / parseInt(splitted[1]);
163           if (Math.abs((4 / 3) - decimalRatio) < min) {
164             min = Math.abs((4 / 3) - decimalRatio);
165             bestRatio = ratio;
166             setDecimalRatio(decimalRatio);
167           }
168         }
169         setRatio(bestRatio);
170       }
171     }
172   }
```

Both these functions are triggered in useEffect when a camera renders.

We developed a method that captures images using the camera component and saves them to the device's media library. This method and its requirements are located in the *camera.js* file. First the user's media library usage permission has to be enquired. This is done in the same manner as it is done for the camera permission itself. The function uses the camera component to take the picture and saves it with the *Media Library* Component which is also available through expo if the permission is given.

```
189
190   const takePicture = async () => {
191
192     if (hasPermission) {
193
194       const options = {
195         quality: 1,
196         base64: true,
197         exif: false
198       }
199
200       const data=await camera.takePictureAsync(options)
201       setImage(data.uri);
202       console.log(data)
203
204       if(hasMediaLibraryPermission) {
205
206         MediaLibrary.saveToLibraryAsync(data.uri).then(() => {
207           setImage(undefined);
208         });
209       }
210     }
211   }
```

With this function there will be the possibility to take pictures via a button in the task screen given by *alternativetask.js* by exporting the functions and then calling it within the timer.

Sound

In order to gain gamification aspects for our app we decided to add sound to our levels. These levels all combine to stories about normal life experiences and each have a screen where you can read a description of the situation that you are about to perform facial expressions to and a screen that shows the task you are about to do and a camera screen where you can see yourself doing the mentioned task. We decided to use sound to increase the fun factor while reading the description. There either is music playing or an audio where the text is read out loud by an actress that we have collaborated with.
When the level is entered the given audio file gets loaded and played. As soon as you leave the screen again it stops. All functionalities which can be found in the *level.js* file are given by the Audio Component of the *expo-av* Library. To have the right audio file playing that for example matches the given text we have to hand it over in the *contentManager.js*

```
435
436 const schnee5 = {
437     "baseText": "Es laufen drei Kinder mit einem Schlitten an Ihnen vorbei. Die Kinder sind außer sich
438     "highlightedText": "Sie freuen sich mit Ihnen und lächeln.",
439     "task": "Lächeln Sie.",
440     "sound": require("../assets/Uebung5_Der_erste_Schnee.wav")
441 }
442
443 const schnee6 = {
444     "baseText": "Sie erinnern sich noch gut daran, wie Sie selbst als Kind im Schnee gespielt haben. Be
445     "highlightedText": "Sie spitzen die Lippen und bewegen sie nach rechts und links.",
446     "task": "Spitzen sie ihre Lippen und bewegen sie sie nach rechts und links.",
447     "sound": require("../assets/Uebung6_Der_erste_Schnee.wav")
448 }
449
450 const schnee7 = {
451     "baseText": "Sie entscheiden sich für den Heimweg. In der Zwischenzeit ist das Schneien stärker gew
452     "highlightedText": "Sie pusten die Wangen auf und lassen die Luft wieder entweichen.",
453     "task": "Pusten sie ihre Wangen auf und lassen die Luft wieder entweichen.",
454     "sound": require("../assets/Uebung7_Der_erste_Schnee.wav")
455 }
456
457 const schnee8 = {
458     "baseText": "Sie wärmen sich am Kaminfeuer auf und kuscheln sich unter die Decke auf dem Sofa.",
459     "highlightedText": "Entspannen Sie sich...",
460     "task": "Fertig!",
461     "sound": require("../assets/LetzterSatz_Der_erste_Schnee.wav")
```

In the level.js we can use this data afterwards.

```
useEffect(() => {
  setCurrentText(getText());
  setCurrentHighlightedText(getHighlightedText());
  setCurrentAudio(getAudio())
}, []);

useFocusEffect(
  useCallback(() => {
    setCurrentText(getText());
    setCurrentHighlightedText(getHighlightedText());
    setCurrentAudio(getAudio());
    playSound();
  }, [])
);
```

To make the sound accessible we need a useState constant and save the audio file with its given path. Then we are able to asynchronously load and play. Also by unloading it in the same manner we are able to stop it.

```
const [sound, setSound] = useState();

async function playSound() {
  console.log('Loading Sound');
  console.log(getAudio())
  // const { sound } = await Audio.Sound.createAsync( require("../assets/Uebung1_Der_erste_Schnee.wav")
  const { sound } = await Audio.Sound.createAsync(getAudio()
  );
  setSound(sound);

  console.log('Playing Sound');
  await sound.playAsync();
}

async function stopSound() {
  await sound.unloadAsync();
}
```

To actually load and play the sounds we call our function via UseEffects.

```
useEffect(() => {
  setCurrentText(getText());
  setCurrentHighlightedText(getHighlightedText());
  setCurrentAudio(getAudio())
}, []);

useFocusEffect(
  useCallback(() => {
    setCurrentText(getText());
    setCurrentHighlightedText(getHighlightedText());
    setCurrentAudio(getAudio());
    playSound();
  }, [])
);


useEffect(() => {
  return sound
    ? () => {
      console.log('Unloading Sound');
      sound.unloadAsync();
    }
    : undefined;
}, [sound]);

useEffect(() => {
  playSound();
}, [])
```

Notifications

notifications.js:

Disclaimer: This function is not implemented!

We are using the standard Expo push notifications, that work both on *Android* and *iOS*. They are set up to activate two times a day, at 11:00 and 18:00 every day and can be disabled.

```
109   //Schedules one or several notifications
110   async function schedulePushMorningNotification(time) {
111     await Notifications.scheduleNotificationAsync({
112       identifier: "one",
113       content: {
114         title: "EmotionAI",
115         body: "Es ist Zeit für eine Übung!",
116         data: { data: "tba" },
117       },
118       trigger: {
119         //seconds: 5,
120         hour: 11,
121         minute: 0,
122         repeats: true,
123       },
124     });
125   }
```

On clicking the notification, the user is brought to the main screen (***menu.js***) where they can immediately continue their exercises.

Dark Mode

In order to switch between light and dark mode, React Native's *useColorScheme* hook is used throughout the app. This hook returns *light* or *dark* (depending on the system's settings) and is assigned to the *colorScheme* constant.

```
const colorScheme = useColorScheme();
```

Whenever a component has a separate look for light and dark mode, *colorScheme* is used to assign the correct color.

```
const containerColor = colorScheme === 'light' ? light_background_color :
dark_background_color;
<View style={{ backgroundColor: containerColor }}></View>
```

Components

button.js

<Button> is a custom component that can navigate between pages, based on React Native's *TouchableOpacity* component. Use the *label* property to change the text inside the button. The *target* property defines which page the button navigates to. The *navigation* property is solely passed for functionality purposes and should not be changed!

Example:

```
<Button label="your label" navigation={navigation} target={"your page"} />
```

customButton.js

<CustomButton> is a custom Component that is implemented with the *Animated* Library. You can pass text, color and onPress props which allows you to customize it so it feeds your needs for a button. Its purpose is to increase the fun factor by bringing some life to the levels by morphing from a Circle to somewhat a square.

exercise.js

<Exercise> is a custom component that navigates to an exercise and is based on React Native's *TouchableOpacity* component. It is usually rendered as part of its parent component <Scenario>. Depending on the scenario the *icon* property and the icon inside <Exercise> changes. Since an exercise can be tagged as locked or unlocked, and completed or not completed, there are four different child components that can be rendered.

<LockedExercise> renders the <Exercise> component with a lock symbol to indicate that an exercise is locked. In this case, the component can not be pressed.

<UnlockedExercise> renders the <Exercise> component without a lock symbol to indicate that an exercise has been unlocked.

<CompletedExercise> renders the <Exercise> component with a check mark to indicate that an exercise has been completed.

<ContinueExercise> renders the <Exercise> component specifically for the “Continue Scenario” section in *menu.js*.

nav_bar.js

Renders a <NavBar> with a page’s title (which is passed through the *page_title* property) at the top of a page.

Example:

```
<NavBar page_title="your page title" />
```

progress_bar.js

Renders a <ProgressBar> that calculates the percentage of exercises completed for a given scenario based on the *exercises* and *progress* properties.

1. 100 is divided by the number of exercises.

```
const x = 100 / props.exercises;
```

2. x is then multiplied by the number of completed exercises and doubled for displaying purposes.

```
const y = x * props.progress * 2;
```

3. The result is passed to the width property.

```
<View style={[{ width: y }, { height: 8 }, { borderRadius: 4 }, progressColor]}></View>
```

For example, you would like to render a progress bar that is filled by 50%:

```
<ProgressBar exercises={2} progress={1} />
```

scenario.js

<Scenario> is a parent component of <Exercise> and <ProgressBar>. It contains every exercise of a given scenario and the user’s progress for a scenario. The Content Managers *getScenario* and *getScenarioLength* functions are used to render the right amount of <Exercise> components, to get a scenario’s title and progress which is then passed to <ProgressBar>.

selection.js

<Selection> lets you choose from several options that are defined in the *option* property.

```
<Selection option1="first option" option2="second option" option3="third option" />
```

By default none of the options are selected and the state variables are set to false.

```
const [optionIsEnabled1, setOptionIsEnabled1] = useState(false);
const [optionIsEnabled2, setOptionIsEnabled2] = useState(false);
const [optionIsEnabled3, setOptionIsEnabled3] = useState(false);
```

When selecting an option the respective series of functions will be called. For example, you select option 1:

```
const selectOption1 = () => [setOptionIsEnabled1(true), setOptionIsEnabled2(false),
setOptionIsEnabled3(false)];
```

As a result, *optionIsEnabled1* is set to *true* while *optionIsEnabled2* and *optionIsEnabled3* are set to *false*.

styles.js

Contains colors and styles used throughout the app.

```
const light_primary_color = 'rgb(0, 122, 255)';
const dark_primary_color = 'rgb(10, 132, 255)';
const light_background_color = 'rgb(255, 255, 255)';
const dark_background_color = 'rgb(0, 0, 0)';
export { light_primary_color, dark_primary_color, light_background_color,
dark_background_color };
```

light_primary_color is the accent color used to highlight buttons, the progress bar, icons, or important text (when dark mode is disabled).

dark_primary_color is the accent color used to highlight buttons, the progress bar, icons, or important text (when dark mode is enabled).

light_background_color is the background color when dark mode is disabled.

dark_background_color is the background color when dark mode is enabled.

tab_bar.js

Renders a `<TabBar>` with icons that indicate which page the user is currently on and can be pressed to navigate between pages. The icons are imported from the FontAwesome library.

```
import { FontAwesomeIcon } from '@fortawesome/react-native-fontawesome';
import { faGear, faHouse } from '@fortawesome/free-solid-svg-icons';
```

Whether an icon is displayed as active or inactive depends on the properties of `<TabBar>`. For example, you want the `<TabBar>` to show that the user is on the menu page:

```
<TabBar home={activeIconColor} settings={inactiveIconColor} navigation={navigation} />
```

The *navigation* property is solely passed for functionality purposes and should not be changed!

Pages

home.js

Renders the first page the user encounters when starting the app. It contains the mimical <Logo> and <Button> components that navigate to the *sign in* page, *sign up* page, or skip and go to the *overview* page.

menu.js

Renders the *overview* page. It contains a <NavBar>, the filter, an <Exercise> component to continue the last scenario that has been played, a <Scenario> component for each scenario, and a <TabBar>.

settings.js

On this page users set their preferences. The default settings are defined in the *config* object.

```
let config = {
  language: "german",
  large Font: false,
  fontSize: 17,
  camera: true,
  notifications: false,
  text: true,
  narrator: true,
  music: true
}
```

When pressing on a <Switch> (which is a React Native component), a series of functions will be called, depending on which option a user wants to change.

For example, the user wants to change the language from german to english:

```
const toggleSwitch1 = () => [
  setIsEnabled1(previousState => !previousState),
  setLanguage(isEnabled1 ? "german" : "english"),
  storeLanguageData(isEnabled1 ? "german" : "english")
 ];
```

In this case, *setIsEnabled1* changes the state variable *isEnabled1* from *true* to *false* (or the other way around) to indicate the change of an option. *setLanguage* changes from *german* to *english* (or the other way around), depending on isEnabled1. *storeLanguageData* stores the selected language in *AsyncStorage* by changing the config object and saving the new config object in AsyncStorage.

```
const storeLanguageData = async (value) => {
  try {
    config.language = value;
    await AsyncStorage.setItem('settings', JSON.stringify(config));
```

```
  } catch (error) {
    // error storing data
  }
}
```

If a component needs to access the language data in AsyncStorage, the *getData* function is used.

```
const getData = async () => {
  try {
    const jsonValue = await AsyncStorage.getItem('settings');
    const value = JSON.parse(jsonValue);
    if (value !== null) {
      setLanguage(value.language);
    }
  } catch (error) {
    // error retrieving data
  }
}
```

After retrieving the config object from AsyncStorage, the component's *language state variable* is updated by *setLanguage*. Now the component can switch between german and english text.

```
<Text>{language == "german" ? "Hallo" : "Hello"}</Text>
```

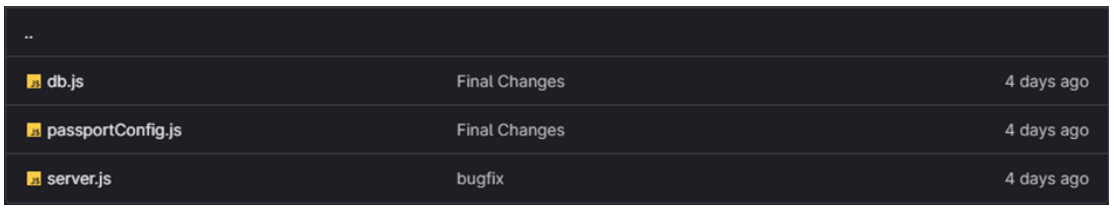
Website

Backend

The Backend folder is divided into three JavaScript files:

- *db.js*, which all establishes a connection to the database.
- *passportConfig.js*, which makes a query to the database and ensures that a user logs in with correct user credentials. If this is the case, the user is serialized and can be logged in.

- *server.js*, which starts the NextJS, as well as the backend server. Here you can also find all post and get requests that are made on the website.



In *db.js* “mysql” and “dotenv” are imported.
“mysql” is pretty much self explanatory since this is the database we are using.
“dotenv” is used to read the .process.env where the login credentials for the database are stored.

```
// Description: This file is used to connect to the database

const mysql = require('mysql');
const dotenv = require('dotenv');


// mySQL password masking
dotenv.config({ path: '.process.env' });

// Create connection to database
var db = mysql.createConnection({
    host: process.env.MYSQL_HOST,
    user: process.env.MYSQL_USER,
    password: process.env.MYSQL_PASSWORD,
    database: process.env.MYSQL_DATABASE
});

module.exports = db;
```

Initially, we import all the important modules and specify that NextJS is started by the backend.

We are also importing several modules and dependencies, including NextJS, Express, Body-Parser, Bcrypt, Cors, Passport, Express-Session, Cookie-Parser.

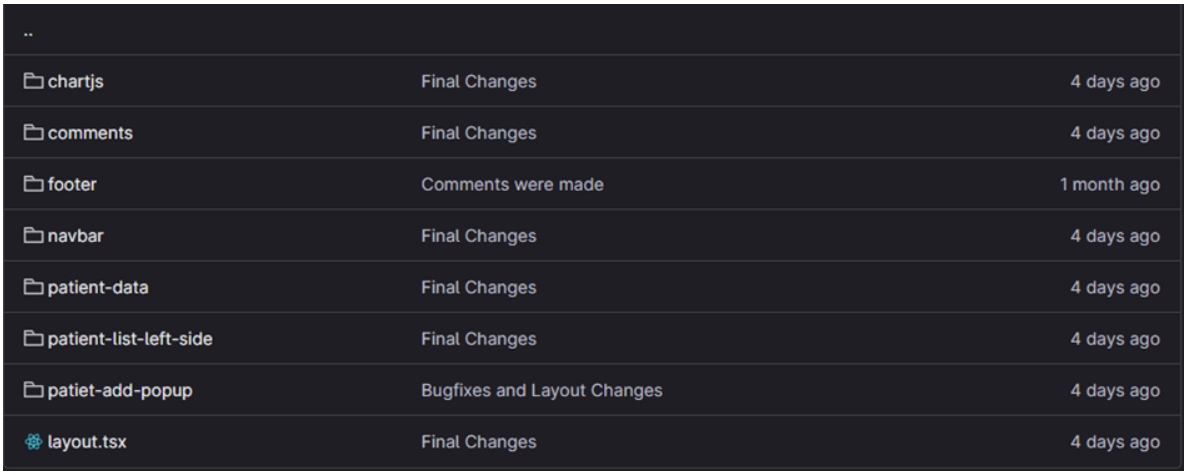
```
// FUNCTIONS:
// Import Next and Express as a requirement for the backend
// BodyParser is used to parse the body of the request
const nextJS = require('next');
const express = require('express');
const bodyParser = require('body-parser');
const bcrypt = require('bcrypt');
const cors = require('cors');
const passport = require('passport');
const expressSession = require('express-session');
const cookieParser = require('cookie-parser');
const cookie = require('js-cookie');
const db = require('./db');
const mysqlStore = require('express-mysql-session')(expressSession);

const dev = process.env.NODE_ENV !== 'production';

// Start NextJS through Express
const app = nextJS({ dev, hostname: 'localhost', port: 3000 });
const handle = app.getRequestHandler();
```

Body-Parser for handling incoming JSON requests, Cors for handling cross-origin resource sharing, Bcrypt for hashing passwords and Cookie-Parser for handling cookies.
The server is listening on port 3000 and whenever a request is made it routes the request to NextJS.

From line 91 on, post and get requests are handled.

Components

The components Folder contains several components which are used inside the project.



chartjs

chartjs stores several charts which are used to display progress patients have made so far. The component starts by importing the necessary modules and libraries.

```
import React, { use } from "react";
import Chart from "chart.js/auto";
import { CategoryScale } from "chart.js";
Chart.register(CategoryScale);
import { Line, Bar } from "react-chartjs-2";
import { useEffect, useState } from "react";
import Axios from "axios";
import { useRouter } from "next/router";
```

After the UseEffect call in which data is retrieved from the database the response is saved in variables. One for each month as well as how many Upper-Face exercises and Lower-Face exercises were made.

```
const charts = () => {
  var [january, setJanuary] = useState([]);
  var [february, setFebruary] = useState([]);
  var [march, setMarch] = useState([]);
  var [april, setApril] = useState([]);
  var [may, setMay] = useState([]);
  var [june, setJune] = useState([]);
  var [july, setJuly] = useState([]);
  var [august, setAugust] = useState([]);
  var [september, setSeptember] = useState([]);
  var [october, setOctober] = useState([]);
  var [november, setNovember] = useState([]);
  var [december, setDecember] = useState([]);

  var [upper, setUpper] = useState([]);
  var [lower, setLower] = useState([]);

  var router = useRouter();
  var index = router.query.index;

  useEffect(() => {
    // We need to check if the router is ready before we can access the query
    if (!router) return;
    // We need to check if the query is ready before we can access the index
    if (!index) return;
    getCharts(index);
  }, [router, index]);

  const getCharts = async (index: any) => {
    try {
      await Axios.post("/api/get-chart-data", {
        index: index,
      }).then((response) => {
        setJanuary(response.data[0]);
        setFebruary(response.data[1]);
        setMarch(response.data[2]);
        setApril(response.data[3]);
        setMay(response.data[4]);
        setJune(response.data[5]);
        setJuly(response.data[6]);
        setAugust(response.data[7]);
        setSeptember(response.data[8]);
        setOctober(response.data[9]);
        setNovember(response.data[10]);
        setDecember(response.data[11]);
        setUpper(response.data[12]);
        setLower(response.data[13]);
      });
    } catch (error: any) {
      error;
    }
```

The component then renders several charts, including a line chart that shows the frequency of exercises done over the course of the year and bar chart that shows how much Upper-Face or Lower-Face is trained.

comments

comments display comments for a certain patient.

It starts by importing the necessary modules and libraries.
After fetching comments from the mysql database, the comments are stored using *useState*.

```
import React from "react";
import { useState, useEffect } from "react";
import { useRouter } from "next/router";
import Axios from "axios";

const comment = () => {
  var [comments, setComments] = useState([]);
  var router = useRouter();
  var index = router.query.index;

  useEffect(() => {
    if (!router.isReady) return; // wait for router to be ready
    // Send a request to the server to get the comments
    try {
      Axios.post("/api/get-comments", { index: index }).then((res) => {
        setComments(res.data);
      });
    } catch (error: any) {
      console.log("Es konnten keine Patienten geladen werden");
    }
  }, [router.isReady]);
```

The component then maps over the comments array and displays each comment along with a delete button for each comment.

The delete button is wired to a deleteCommentHandler function that when clicked, the comment will be deleted. When a comment is deleted, the page is refreshed by calling the refreshPage() function. The component also includes a dateTimeHelper function that takes in a date and returns a formatted date and time. This helper function is used to display the date and time of when the comment was added.

footer

The footer is not used.

navbar

The component starts by importing the necessary modules and libraries.

```
import React, { useState, useEffect } from "react";
import Link from "next/link";
import axios from "axios";
import { useRouter } from "next/router";
import Image from "next/image";
import AppLogo from "../../public/mimical_logo.svg";
```

The navbar checks if the current route (via useRouter()) is != login.

```
  const router = useRouter();
  const currentRoute = router.pathname;
  const currentlogin = router.pathname;
```

```
if (currentlogin !== "/login") {
```

If this is the case the navbar will be displayed. This has the ulterior motive that you do not have to include an additional function for the login page. If the current route is /login, the navbar is not displayed.

The navbar has a logo, a title, and a link to logout. When the logout link is clicked, the component sends a post request to the server.

```
  const logout = () => {
    axios.post("/api/logout").then((res) => {
      refreshPage();
    });
  };
```

Once the logout is successful, the component calls the refreshPage() function to reload the page.

The logout post also ensures that the cookie is removed locally and from the session in the database. This makes it impossible to re-enter the dashboard.

patient-data

patient-data ensures that the patient data of the respective patients is loaded. This includes personal data as well as diagnoses.

The component starts by importing the necessary modules and libraries.

```
import Axios from "axios";
import Image from "next/image";
import { useRouter } from "next/router";
import React, { useEffect, useState } from "react";
import Popup from "reactjs-popup";
import ProPic from "../../public/istockphoto-1223671392-612x612.jpg";
```

Patient Data has several variables.

In “patients” the patients data received from the backend is stored.

```
const index = () => {
  var router = useRouter();
  var index = router.query.index;
  var [open, setOpen] = useState(false);

  var [patients, setPatients] = useState([]);
  var [error, setError] = useState(false);
  var [diagnose, setDiagnose] = useState("");
  var [diagnoseChanged, setDiagnoseChanged] = useState(false);
  var [diagnoseSend, setDiagnoseSend] = useState("");
  var [interests, setInterests] = useState("");
  var [interestsChanged, setInterestsChanged] = useState(false);
  var [interestsSend, setInterestsSend] = useState("");
  var [affectedSide, setAffectedSide] = useState("");
  var [affectedSideChanged, setAffectedSideChanged] = useState(false);
  var [affectedSideSend, setAffectedSideSend] = useState("");
  var [numbness, setNumbness] = useState("");
  var [numbnessChanged, setNumbnessChanged] = useState(false);
  var [numbnessSend, setNumbnessSend] = useState("");
  var [motion, setMotion] = useState("");
  var [motionChanged, setMotionChanged] = useState(false);
  var [motionSend, setMotionSend] = useState("");
```

The next section is applicable to all other variables.

In diagnosis, the new diagnosis is saved when editing the patient data. If this has been edited, *diagnoseChanged* is set to true. This has the background that the system knows

which data must be transferred and which remain the same. In an If query it is now checked whether something has changed.

```
// Checks if the therapist changed data of a patient
const isDataChanged = () => {
  if (diagnoseChanged) {
    diagnoseSend = diagnose;
  } else {
    diagnoseSend = patients
      .filter((patient: any) => patient.ID == index)
      .map((patient: any) => patient.diagnose)
      .toString();
  }

  if (interestsChanged) {
    interestsSend = interests;
  } else {
    interestsSend = patients
      .filter((patient: any) => patient.ID == index)
      .map((patient: any) => patient.interests)
      .toString();
  }
```

The patient-data is first mapped by the ID and then mapped onto the screen in a grid layout.

```
          patients
            .filter((patient: any) => patient.ID == index)
            .map((patient: any) => (
              <div
                key={patient.ID}
                className="m-5 grid grid-rows-6 grid-flow-col gap-y-0.5 gap-x-3"
              >
                <div className="text-sm">
                  Name: {patient.prename} {patient.name}
                </div>
                <div className="text-sm">Patienten ID: {patient.ID}</div>
                <div className="text-sm">
                  Geschlecht: {genderChecker(patient.gender)}
                </div>
                <div className="text-sm">
                  Geburtsdatum : {getBirthdate(patient.birthdate)}
                </div>
                <div className="text-sm">
                  Taubheitsgefühl?: {patient.numbness}
                </div>
                <div className="text-sm">Diagnose: {patient.diagnose}</div>

                <div className="text-sm">Email: {patient.email}</div>
                <div className="text-sm"></div>
                <div className="text-sm"></div>
                <div className="text-sm">
```

The “Bearbeiten” button in line 167 is used to open a popup window in which you can change patient data.

patient-list-left-side

patient-list-left-side loads all the patients which are assigned to a specific therapist.

The component starts by importing the necessary modules and libraries.

```
1  import Link from "next/link";
2  import React, { useEffect, useState } from "react";
3  import { useRouter } from "next/router";
4  import axios from "axios"; // axios is a library that allows us to make HTTP requests
5  import Popup from "../patiet-add-popup/popup";
6  import "reactjs-popup/dist/index.css";
```

As in the previous section explained, first patient data is stored in variables.

```
8  const Patientlist = () => {
9    const [patients, setPatients] = useState([]);
10   const [searchTerm, setSearchTerm] = useState("");
11   const [user, setUser] = useState([]);
12
13   const router = useRouter();
14   const currentRoute = router.pathname;
15
16   // useEffect is a React hook that allows us to run code when the component is mounted
17   // axios is used to make HTTP requests
18   useEffect(() => {
19     try {
20       axios.get("/api/patient-data").then((res) => {
21         setPatients(res.data);
22       });
23     } catch (error: any) {
24       console.log(error);
25     }
26   }, []);
27
28   useEffect(() => {
29     axios.get("/api/getUser").then((res) => {
30       setUser(res.data);
31     });
32   }, []);
```

The current therapist logged in will be displayed.

```
<div
  className="pt-4 px-2 pl-2 pr-2 text-xs grid justify-items-center
 transition font-light duration-500 focus:ring-4 focus:outline-none focus:ring-pink-200 dark:focus:ring-pink-800 rounded-full"
>
  Eingeloggt als:
</div>
<div
  className="px-2 pl-2 pr-2 grid justify-items-center
 text-xs transition font-bold duration-500 focus:ring-4 focus:outline-none focus:ring-pink-200 dark:focus:ring-pink-800 rounded-full"
>
  {user[0] + " " + user[2] + " " + user[1]}
</div>
```

The component has a search bar that allows the user to filter the list of patients by name or ID. The search bar uses the onChange event to update the searchTerm state variable with the current search term entered by the user.

```
<div className="bg-black/10 mb-5 w-4/5 rounded-xl">
  {isSearch(searchTerm) ? (
    <div className="m-2 text-xs">
      <div className="grid justify-items-left text-lg">
        {typeof patients === undefined ? (
          <div>loading...</div>
        ) : (
          patients
            .filter((patient: any) => patient.therapistID === user[3])
            .map((patient: any) => (
              <Link key={patient.ID} href={"/patients/" + patient.ID}>
                <div
                  className="text-center
                 custom-blue m-1 rounded-lg text-white"
                >
                  {patient.prename + " " + patient.name}
                </div>
              </Link>
            ))
        )}
      </div>
    </div>
```

If the search bar is empty every patient with

patient-add-popup

patient-list-left-side loads all the patients who are assigned to a specific therapist.

The component starts by importing the necessary modules and libraries.

```
1  import axios from "axios";
2  import React, { useEffect, useState } from "react";
3  import Popup from "reactjs-popup";
```

As in the previous section explained, first patient data is stored in variables.

```
const popup = () => {
  const [patientkey, setPatientkey] = useState("");
  const [error, setError] = useState(false);
  const [open, setOpen] = useState(false);
  const [addedPatientOpen, setAddedPatientOpen] = useState(false);
  const [addPatientData, setAddPatientData] = useState(Object);

  const [diagnose, setDiagnose] = useState("");
  const [interests, setInterests] = useState("");
  const [affectedSide, setAffectedSide] = useState("");
  const [numbness, setNumbness] = useState("");
  const [motion, setMotion] = useState("");

  // useEffect is a React hook that allows us to run code when the component is mounted
  // axios is used to make HTTP requests

  const sendPatientkey = async () => {
    await axios
      .post("/api/check-patient-key-from-dashboard", { key: patientkey })
      .then(async (res) => {
        if (res.data === "nothing") {
          setError(true);
          await sleep(5000);
          setError(false);
        } else {
          setAddPatientData(res.data);
          setAddedPatientOpen(true);
        }
      });
  };

  const addPatient = async () => {
    const data = {
      diagnoseSend: diagnose,
      interestsSend: interests,
      affectedSideSend: affectedSide,
      numbnessSend: numbness,
      motionSend: motion,
      ID: addPatientData.ID,
    };

    try {
      await axios.post("/api/add-patient", data).then((res) => {
        refreshPage();
      });
    } catch (error) {
      console.log("error");
    }
  };
```

When the button "Neuen Patienten Hinzufügen" is clicked, it calls the setOpen() function which causes the first pop-up modal, using the "Popup" component, to be rendered on the

page with a message asking the user to enter a "Patientenschlüssel" (patient key) to register a new patient.

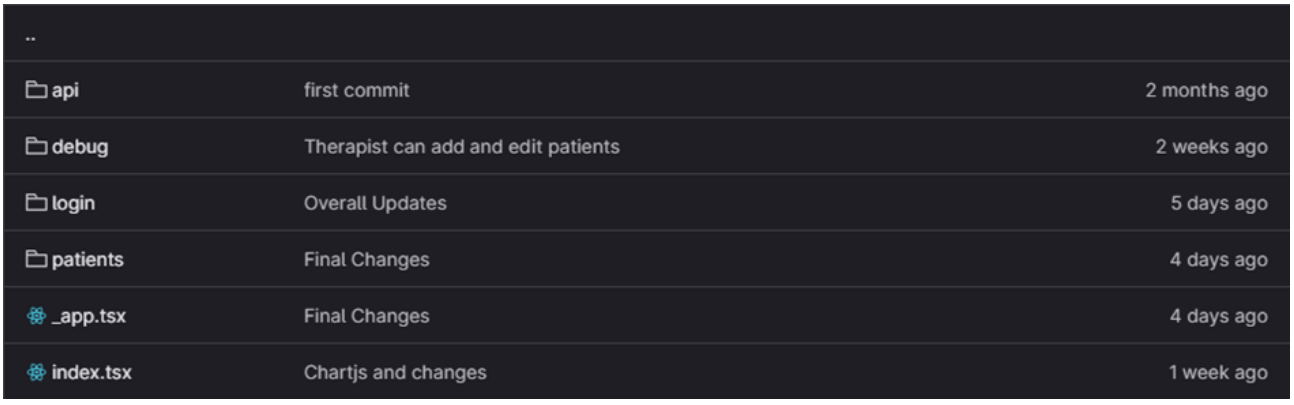
Inside the first modal, there is an input field where the user can enter the patient key, and a "Hinzufügen" (add) button. When the user enters a patient key and clicks the "Hinzufügen" button, the sendPatientkey() function is called.

```
21  const sendPatientkey = async () => {
22    await axios
23      .post("/api/check-patient-key-from-dashboard", { key: patientkey })
24      .then(async (res) => {
25        if (res.data === "nothing") {
26          setError(true);
27          await sleep(5000);
28          setError(false);
29        } else {
30          setAddPatientData(res.data);
31          setAddedPatientOpen(true);
32        }
33      });
34  };
```

If there is an error, such as a patient key that does not match any existing patients, a message will be displayed in the first modal with the text "Es gibt keinen Patienten mit diesem Schlüssel, bitte versuchen Sie es erneut.".

If a new patient is successfully added, the second modal is rendered with the "open" state set to true by the setAddedPatientOpen() function. This modal displays the patient's data, such as first name, last name, ID, and email. The inputs are disabled and filled with the data of the added patient.

Pages



Since only login and patients are relevant here, only these two will be discussed in the following.

login

The login page is quite simple. It contains a centered input field which stores input data onChange() in two strings. If everything is set the handleLogin() function sends a post request to the server.

```
15    const handleLogin = () => {
16      axios({
17        method: "POST",
18        data: {
19          username: email,
20          password: password,
21        },
22        withCredentials: true,
23        url: "http://localhost:3000/api/login",
24      }).then(async (res) => {
25        if (res.data === false) {
26          refreshPage();
27          router.push("/");
28        }
29
30        if (res.data === true) {
31          setError(true);
32          await sleep(5000);
33          setError(false);
34        }
35      });
36    };
```

After authenticating the therapist, a cookie is set and the router pushes the therapist to the dashboard page.

The middleware checks if the cookie corresponds to the session entered in the database.

```
10    let url = req.url;
11    const verify = req.cookies.get("connect.sid");
12
13    if (!verify && !(url === "http://localhost:3000/login")) {
14      if (url.includes("/_next")) {
15        url.match;
16        return NextResponse.next();
17      } else {
18        return NextResponse.redirect("http://localhost:3000/login");
19      }
20    }
21
22    if (verify && url === "http://localhost:3000/login") {
23      if (url.includes("/_next")) {
24        return NextResponse.next();
25      } else {
26        return NextResponse.redirect("http://localhost:3000/");
27      }
28    }
29  }
```

patients

The patients page starts with getting patient-data from a get request and storing them into a patients array. It is important to notice that the get request only sends patient data which is bound to the therapists ID.

```
// Dynamic Patient Page

const index = () => {
  const [patients, setPatients] = useState([]);
  var router = useRouter();
  var index = router.query.index;

  var [addComment, setAddComment] = useState("");
  var [open, setOpen] = useState(false);

  useEffect(() => {
    try {
      Axios.get("/api/patient-data").then((res) => {
        setPatients(res.data);
      });
    } catch (error: any) {
      error;
    }
  }, []);

  const addCommentHandler = () => {
    try {
      Axios.post("/api/add-comment", {
        index: index,
        comment: addComment,
      }).then(() => {
        // setOpen(false);
        refreshPage();
      });
    } catch (error: any) {
      console.log("Es konnten keine Patienten geladen werden");
    }
  };
```

The PatientList component loads the Sidebar and the PatientData component loads the patient data.

To get the right patient, the router checks for the dynamic page number which is reverencing the ID of the patient. This way we can filter for the right patient.

```
  var router = useRouter();
  var index = router.query.index;
```

At the bottom the comments are displayed with the Comments component.

```
            <div>
              <Comments />
            </div>
```

The possibility to add comments is also given by pressing the “Kommentar hinzufügen” button.

middleware

The middleware is an important module, which is used to check if the therapist is verified. By requesting the cookie from the browser we can make sure that an unauthorized user cannot get access to the dashboard. But even if an unauthorized user manages to access the dashboard, the built- in security inside of the backend prevents the user from getting requests in the first place.

```
import { NextResponse } from "next/server";
import type { NextRequest } from "next/server";

export function middleware(req: NextRequest, res: NextResponse) {
  //Get the host from headers
  const host = req.headers.get("host");
  //Remove the port from the host name, e.g. "localhost:3000" -> "localhost"
  const hostname = host?.split(":")[0];

  let url = req.url;
  const verify = req.cookies.get("connect.sid");

  if (!verify && !(url === "http://localhost:3000/login")) {
    if (url.includes("/_next")) {
      url.match;
      return NextResponse.next();
    } else {
      return NextResponse.redirect("http://localhost:3000/login");
    }
  }

  if (verify && url === "http://localhost:3000/login") {
    if (url.includes("/_next")) {
      return NextResponse.next();
    } else {
      return NextResponse.redirect("http://localhost:3000/");
    }
  }
}
```

Database

Adding a database to our project was a critical step in ensuring that we wrote the code to immediately adapt to the database. We experimented with local databases for the time being, but quickly realized that this step was inevitable.
After evaluating various options, we decided to use MySQL as the basis for our database.

To implement the database, we first had to set up a hosting provider. We chose to host our database on strato.de.

After setting up the hosting, we installed MySQL on our server and configured it to meet the specific requirements of our project. This involved creating tables, setting up user accounts, and configuring security settings to ensure the protection of sensitive data.