

Datenbanken und Web-Techniken
Project Task Summer Semester 2024

Name: Jahandar Hakhiyev
Study course: Web Engineering
Matriculation number: 768924

1. Utilized technologies.

For this project I have used the following techniques and tools:

For frontend I used React to build UI, it's hooks like `useState`, `UseContext`, `UserLoaderData`, `UseNavigate`, `React Router` for client side routing, `axios` for making HTTP requests, `React Leaflet` for map and `Sass` for styling the web components.

I chose React because of its adoption and comfortable ecosystem. The wide usage of React is evident in the popularity of libraries like `styled-components` and `react-router-dom` within the React community. In addition, I have previous experience in React development, allowing me to leverage my existing expertise and ensure efficient development and maintenance of our project.

For the server side I used `Express` for web application, `bcrypt` for encryption, `json web token (JWT)` for API authentication, `axios` for making HTTP requests.

Because of its middleware support, I decided to go with `Express`. We can streamline the development process thanks to this functionality, which makes crucial activities like processing requests, logging, and authentication simpler. And once more, it allows me to take advantage of my prior experience with `Express`. As `bcrypt`, `JWT`, and `axios` are widely used tools with a wealth of online information, I have opted to use them.

I utilized `MongoDB` for the database, and built models using `Prisma Schema`. Because of its performance, scalability, and flexibility, `MongoDB` is my choice. Handling complicated data is made easier by its document-based format, and `MongoDB Atlas` provides a fully managed database solution. I choose to utilize `Prisma` because it makes database management and access easier for applications. It offers a setting that is conducive to developers. Three models are kept in the database: `Post`, `SavedPost`, and `User`. All post details are contained in the post model, all user details are contained in the user model, and inside the `SavedPost` model, a relationship between user and post details allows one to determine which user saved which post.

2. Architecture and functionality

The homepage of the application, "edumap", features a clean design aimed at exploring youth services and filtering them by 4 categories(Figure 1). There is a navigation menu with links to "Home", "About" and "Youth Service". When hovering a "Youth Service" additional window becomes visible where we can further filter the results. On the right side, there are options to "Sign in" and "Sign up," which change to "Profile" and profile name when logging in.

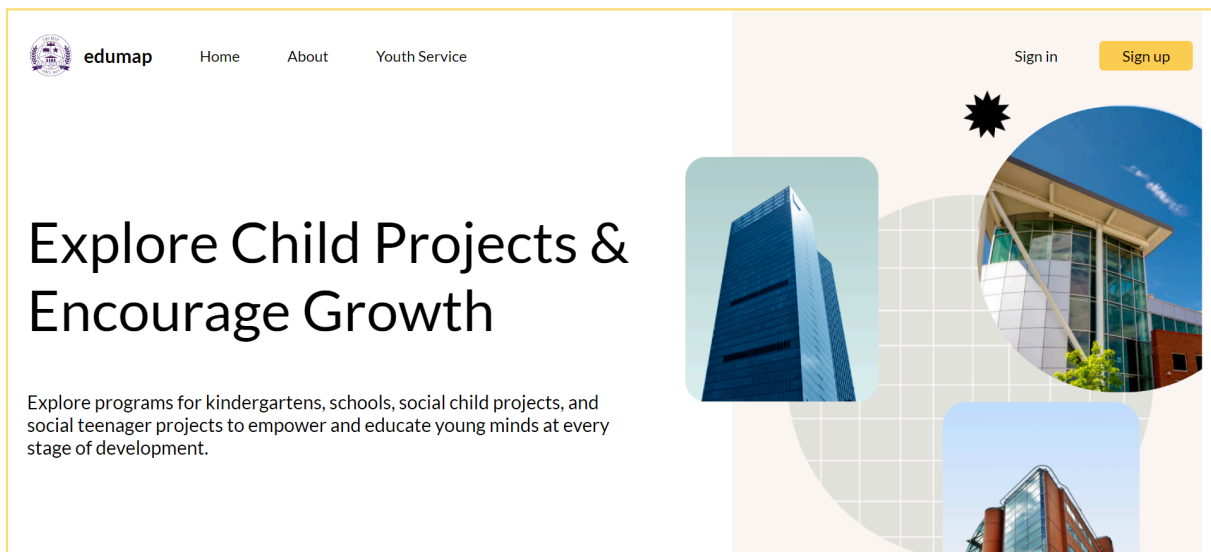


Figure 1: Homepage

On the registration page, users will need to enter their username, email, password and their home address(street and postal code). Once they have entered their account password, the website navigates them into the sign in page. When they sign in, the server sends the token to the browser and saves it in the browser's local storage for up to one week(Figure 2).

Create an Account

Sign up


Do you have an account?

Figure 2: User Registration Page

After signing in, users can enter their profile page. On the profile page, users can view their username, email, and address (Figure 3). They have options to update their account information and log out. After updating their account, their data is automatically updated in the database. If a user clicks to Delete Account button, their information will be removed and they will be redirected to the homepage.

User Information

Update Profile



Username: Jahandar Hakhiyev

email: jahandar.hakhiyev@s2022.tu-chemnitz.de

Address: 09126, Reichenhainer Straße 37

Log out

Delete Account

Figure 3: Profile Page

At the bottom, users can find their saved post. They have to be logged in in order to save a post (Figure 4).

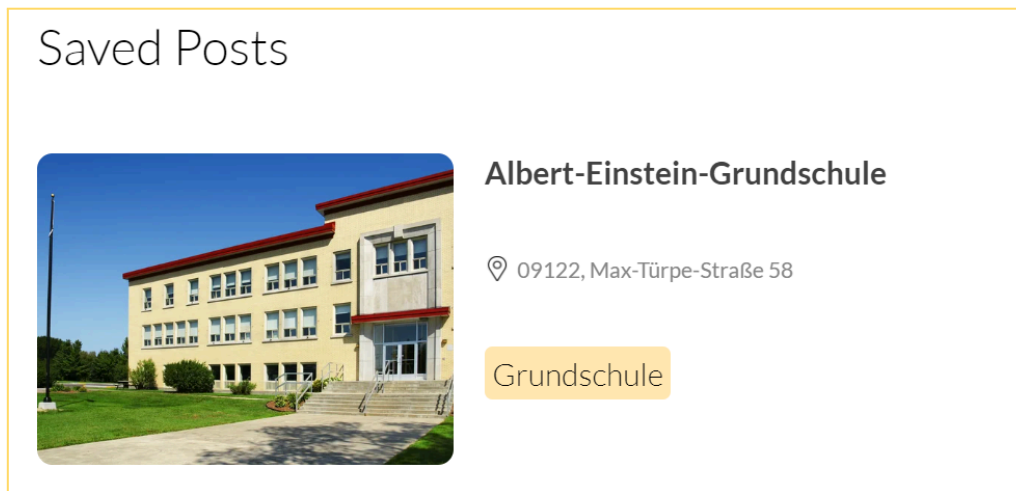


Figure 4: Saved Posts

Users can click it to see the post in a detailed page, and they can delete it from the saved list. Each user can save only one post (Figure 5).

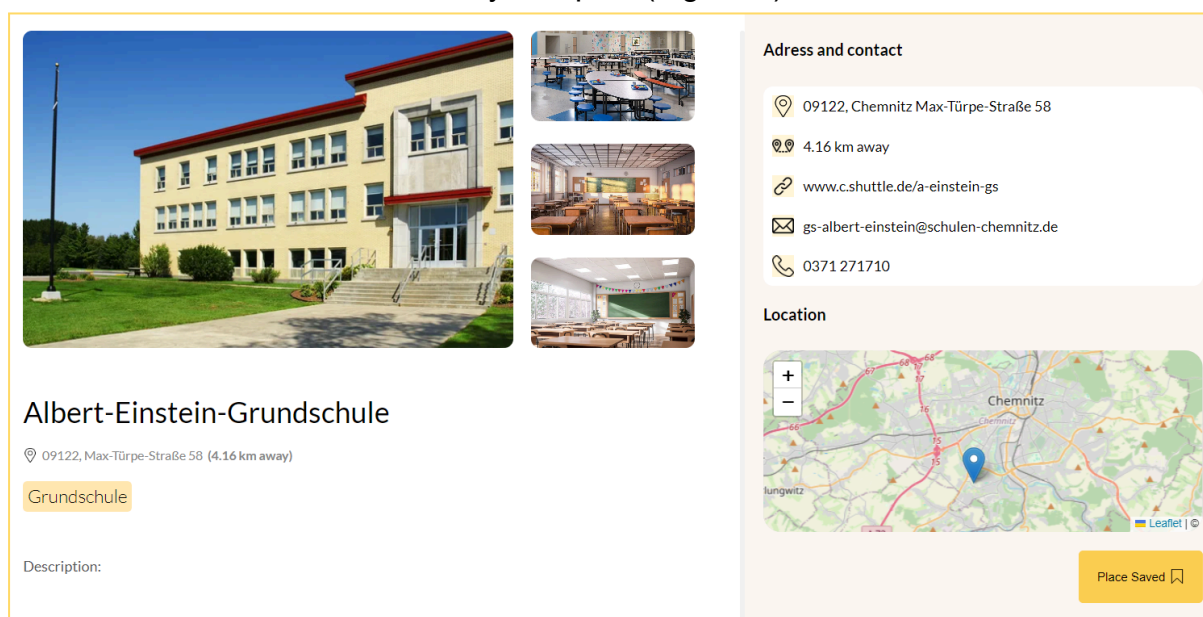


Figure 5: Saved Post in a detailed page


There is a map component where users can see all the places based on their search results, they can click and see the detailed information about the place. When they click the information or the image on the popup, they will be redirected to a related detailed page (Figure 6).



Figure 6: Map component and popup

3. Calculation of distance.

Distance calculations were made using JavaScript. 4 parameters are used in the calculation: longitude and latitude of a user address and facility address (Figure 7). The program first fetches the user data from local storage and parses it into a JavaScript object, then creates a URL for querying the OpenStreetMap Nominatim API to fetch geolocation data based on the constructed address. The calculated distance is rounded to two decimal places and then stored in the component's state using "setDistance".



```

1  useEffect(() => {
2      const handleCalculate = async () => {
3          const user = JSON.parse(localStorage.getItem("user"));
4          if (!user) {
5              setError("User address not found in local storage.");
6              return;
7          }
8
9          const address = user.street + " " + user.plz;
10         const url = 'https://nominatim.openstreetmap.org/search?format=json&q=' + encodeURIComponent(address);
11         const currentCoordinate = post.coordinate;
12
13         try {
14             const response = await fetch(url);
15             const data = await response.json();
16             if (data.length > 0) {
17                 const location = data[0];
18                 const userCoordinate = { lat: parseFloat(location.lat), lng: parseFloat(location.lon) };
19                 const distance = haversineDistance(userCoordinate, currentCoordinate);
20                 setDistance(distance.toFixed(2));
21             } else {
22                 setError("No results found");
23             }
24         } catch (error) {
25             console.error("Error fetching location:", error);
26             setError("An error occurred while fetching location data.");
27         }
28     };
29
30     handleCalculate();
31 }, [post]);

```

Figure 7: Function to calculate the distance.

4. Hashing password

The following code snippet demonstrates how to hash a password using bcrypt:

```

import bcrypt from "bcrypt"
const hashedPassword = bcrypt.hash(password, 10);

```

In this example, “bcrypt.hash(password, 10);” hashes the plain text password with a salt factor of 10. I used this function when a user registers and sets a password, and updates the password.

5. Web Service API

5.1.1. User Registration

Endpoint: /api/auth/register, HTTP Method: POST

Request parameters

Username: String

Email: String

Street: String

Plz: String

Password: String

Response

- Server error. Status code: 500
- Register successfully. Status code: 201

5.1.2. User Login

Endpoint: /api/auth/login, HTTP Method: POST

Request parameters

Username: String

Password: String

Response

- If the username is not found in the database. Status code: 401, message: "Invalid credentials"
- Successful login. Status code: 200
- Server error. Status code: 500

5.1.3. User Logout

Endpoint: /api/auth/logout, HTTP Method: POST

Response

- Successful logout. Status code: 200

5.2.1. User Update Profile

Endpoint: /api/users/:id, HTTP Method: PUT, headers: {'Content-Type: 'application/json'}

Request Parameters

username: String

email: String

password: String
street: String
plz: String

Response

- if ids do not match. Status code: 403, message: "not authorized"
- Success. Status code: 200
- Server error. Status code: 500

5.2.2. Delete user profile

Endpoint: /api/users/:id, HTTP Method: DELETE

Response

- if ids do not match. Status code: 403, message: "not authorized"
- Success. Status code: 200
- Server error. Status code: 500

5.3.1. Save Post

Endpoint: /api/users/save, HTTP Method: POST

Request body

PostId: String
UserId: String

Response

- If the facility already exists in the database. Status code: 200, message: "Post removed"
- If a user saved another facility before. Status code: 403, message: "You cannot save more than one post"
- Success. Status code: 200
- Server error. Status code: 500

5.4.1. Get all schools

Endpoint: /api/posts/school, HTTP Method: GET

Response

- Success. Status code: 200
- Server error. Status code: 500

5.4.2. Get all kindergardens

Endpoint: /api/posts/kindergarden, HTTP Method: GET

Response

- Success. Status code: 200
- Server error. Status code: 500

5.4.3. Get all social child projects

Endpoint: /api/posts/social-child, HTTP Method: GET

Response

- Success. Status code: 200
- Server error. Status code: 500

5.4.4. Get all social teenager projects

Endpoint: /api/posts/social-teenager, HTTP Method: GET

Response

- Success. Status code: 200
- Server error. Status code: 500

5.4.5. Get single school

Endpoint: /api/posts/school/:id, HTTP Method: GET

headers: {
 'Content-Type': 'application/json'
}

Response

- Success. Status code: 200
- Server error. Status code: 500

5.4.6. Get single kindergarden

Endpoint: /api/posts/kindergarden/:id, HTTP Method: GET

headers: {
 'Content-Type': 'application/json'
}

Response

- Success. Status code: 200
- Server error. Status code: 500

5.4.7. Get single social child project

Endpoint: /api/posts/social-child/:id, HTTP Method: GET

headers: {
 'Content-Type': 'application/json'

```
}
```

Response

- Success. Status code: 200
- Server error. Status code: 500

5.4.8. Get single social teenager project

Endpoint: /api/posts/social-teenager/:id, HTTP Method: GET

headers: {

 'Content-Type': 'application/json'

}

Response

- Success. Status code: 200
- Server error. Status code: 500

5.5. JWT authentication

- If there is no token. Status code: 401, message: "not authenticated"
- If the token is expired or is not valid. Status code: 403, message: "token is not valid"

6. Database models

User model

Entities:

- id - String
- email - String (unique)
- username - String (unique)
- password - String
- avatar - String (nullable)
- createdAt - DateTime
- street - String (nullable)
- house No - String (nullable)
- plz - String (nullable)

Relation:

SavedPosts - SavedPost[]

Post model

Entities:

- id - String
- typeOf - String
- title - String (nullable)
- type - String (nullable)
- description - String (nullable)
- sponsor - String (nullable)
- email - String (nullable)
- phoneNumber - String (nullable)
- website - String (nullable)
- language - String (nullable)
- createdAt - DateTime
- street - String (nullable)
- plz - String (nullable)
- coordinate - Array (nullable)
- streetNo - String (nullable)

Relation:

SavedPosts - SavedPost[]

SavedPost model

Entities:

id String
userId String
postId String

Relations:

user - User, fields: userId, reference: id
post - Post, fields: postId, reference: id

7. Fetching data and saving it into database

When a server starts running, it pulls public data from multiple APIs and saves it on a database. It fetches data related to schools, kindergartens, social child services, and teenager services from public APIs, processes the data, and stores it in a MongoDB database using Prisma ORM. For each dataset (school, kindergarten, social child services, and social teenager services), HTTP GET requests are made to the public API endpoints using Axios. However, there are no separate four models in the database. Instead, there is one mode, and inside each model there is "TypeOf" and its values are correspondingly "school", "kindergarden", "child", and "teenager"

Firstly, a GET request is sent to the API endpoint for school data. The response contains a list of features, each representing a school. For each feature, the relevant properties are extracted and stored in the database using Prisma(Figure 8). Similar processes are followed for kindergartens, social child services, and social teenager services.

```

async function fetchSchool() {
  try {
    const response = await axios.get('https://services6.arcgis.com/
    jiszdsDupTU0BfSM/arcgis/rest/services/
    Schulen_OpenData/FeatureServer/0/query?outFields=*&where=1%3D1&f=geojson');
    const data = response.data.features;

    for (const item of data) {
      await prisma.post.create({
        data: {
          typeOf: "school",
          title: item.properties.BEZEICHNUNG,
          type: item.properties.ART,
          description: item.properties.BEZEICHNUNGZUSATZ,
          email: item.properties.EMAIL,
          phoneNumber: item.properties.TELEFON,
          website: item.properties.WWW,
          language: item.properties.SPRACHEN,
          street: item.properties.STRASSE,
          plz: item.properties.PLZ,
          coordinate: item.geometry.coordinates
        },
      });
    }

    console.log('Data successfully saved to database.');
```

```

  } catch (error) {
    console.error('Error fetching or saving data:', error);
  } finally {
    await prisma.$disconnect();
  }
}

```

Figure 8: Fetching and saving process for Schools

Appendix

Reference of utilized technologies

- React - <https://react.dev/>
- Axios - <https://axios-http.com/>
- Leaflet - <https://leafletjs.com/>
- React leaflet - <https://react-leaflet.js.org/>
- React Router - <https://reactrouter.com/>
- Express - <https://expressjs.com/>
- MongoDB - <https://www.mongodb.com/>
- Prisma - <https://www.prisma.io/>
- Node.bcrypt.js - <https://github.com/kelektiv/node.bcrypt.js/>
- Json web token - <https://github.com/auth0/node-jsonwebtoken/>
- Icons - <https://www.flaticon.com/>
- Nominatim - <https://nominatim.org/>
- Sass - <https://sass-lang.com/>