Map Your Run

Group ID: 1207

Members: Philipp John, Rebecca Hartmann, Anja Bürki

Language: JavaScript

Link to App: https://map-your-runs.vercel.app/

Course: Skills – Programming with Advanced Computer Languages

Professor: Prof. Dr. Mario Silic

Submission Date: December 22, 2022

Summary: Next.js-based web application that visualizes geo-based sports activities from Strava on a dashboard and heatmap

---

1) Project Description

This is a mandatory group project which is part of the course: *Skills: Programming with Advanced Computer Languages*.

Our project is written in JavaScript and uses the popular react-based framework Next.js.

In this project, we focused on developing two features which both deal with analyzing our (Philipp's) own physical sports activities such as running, cycling, or skiing:

- Heatmap: Display sports activities on a heatmap to visualizes the locations where Philipp did sports
- Dashboard: Display two analytics charts – The first chart counts the activities by sports type during the selected year, the second chart calculates the distance of the selected activity type over the time

Both features, as well as the main app, are implemented JavaScript. Additionally, the user can filter different types of sports activities if the user only wants to see specific activity types he undertook (e.g., only runs), and can filter during what year the activities should have taken place (e.g., only 2020). Additionally, the app counts how many times a user did which sport activity within a specified year and counts the covered distance by physical activity.

To implement this project, we first had to downloaded Philipp's sports activity data measurements which were measured through a digital device like an Apple Watch.

So far, a static data file has been used for the application. In the future, we could improve the application by allowing anyone to upload his or her own sports data, and even accessing user data through APIs of popular sports aggregation platforms such as Garmin or Strava. Therefore, our application would display a dashboard and heatmap customized to any user's data.

2) How to Run

Visit **https://map-your-runs.vercel.app/** for a deployed demo of this app on Vercel.

To run locally, first clone the repository, install all dependencies, and run the development server from the root directory:

```
npm run dev
# or
yarn dev
```

Open http://localhost:3000 with your browser to see the result.

3) How to Use

The web application "**Map Your Run**" can be accessed through any web browser and has developed to be responsive so that it can be used either on a laptop, tablet, or smartphone.

When starting the web application, the user finds himself on a landing page. This page is here to select how the user wants to see his sporting activities. Currently the possibilities to choose are "Heatmap" and "Dashboard". This landing page serves to provide a space where further features (once developed) could get displayed.

After clicking on the category "**Heatmap**" the user is redirected to the site where he can see on a world map where Philipp has exercised. A user can simply navigate the world map by clicking and dragging it. It is also possible to zoom into the map to get more detailed information of the users sporting activities in a certain region. Another useful feature is the filter option. Currently there are two options the user can filter for:

- The first filter enables the user to filter different sports activity types. This opportunity gives the user a more flexible possibility to discover in which sports he exercises the most (indicated by a red tone on the heatmap) and where he practices certain sports the most (indicated by a larger and higher number of dots on the heatmap)
- The second filter allows to select different years. This feature lets the user see his development of sporting activities over the years. In the following chapter the code underlying these features is displayed in a commented format. The sub-headline displays a warning to the user if the current filter selection (types and years) yields an empty selection, and no results on the heatmap.

After clicking on the category "**Dashboard**" the user is redirected to the site where he can see the two graphs which calculate the activities.

- The first graph counts all activities by type during the specific year the user selected. The filter allows the user to filter by every year from which he has data. The bar chart then shows, in descending order, the number of times the respective sport was performed.
- The second graph calculates the covered distance for each activity over time. The filter enables the user to filter by the specific sports activity.

---

4) Code Descriptions

The following code descriptions serve to provde an overview of the most important components. For an exhaustive view on the components, please view the full source code in the GitHub repository.

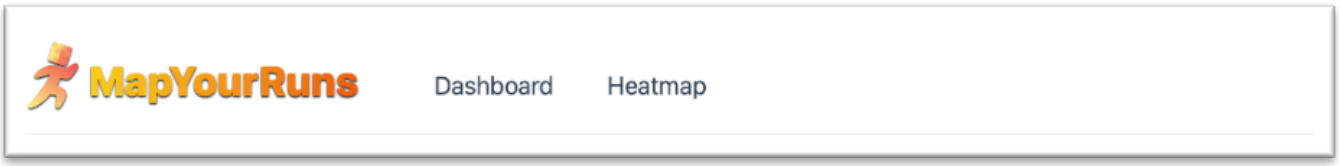
4.1) Gathering Data

Before developing the main app, we had to collect our data. The datapoints to visualize the activities on the heatmap is collected from a Philipp's personal sports data on Strava. The data is collected via Postman and accessible through an API provided by Strava. Thereby the data was exported as a `.json` file and converted to a `.geojson` with the usage of a Python script (can be found in `utils/jsonToGeojson.py`). The geojson was then filtered to only contains datapoints with geolocations. The final dataset consists of 924 activities recorded in a time span from 2014 to 2022. The data holds information about each activity that Philipp has performed, including what type of activity it was (e.g., Run, Ride, Hike, Snowboard, etc.), when it was performed (date), what the tracked GPS-points were (a polyline which can be converted into latitude and longitude points for coordinates), and much more (calories, heartrate, distance, etc.).

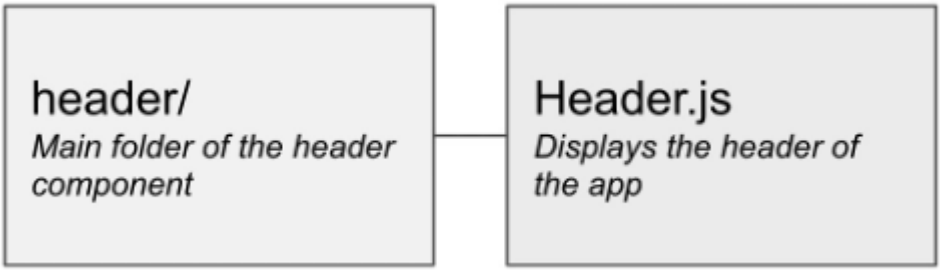
4.2) Header Component

The components `Header` and `Home` each consist of one file.

Component Preview:



Component Graph:



Description:

The file in the component of the `Header` ensures the functioning of the header bar of the web application. The header is kept rather simple and allows to navigate to the different websites (home, dashboard and heatmap). To link the different sites, `NavLinks` were implemented:

```
<NavLink href="/dashboard">Dashboard</NavLink>
<NavLink href="/heatmap">Heatmap</NavLink>
```

Depending on the screen size, the Header changes from a desktop version to a mobile version using Tailwind css properties:

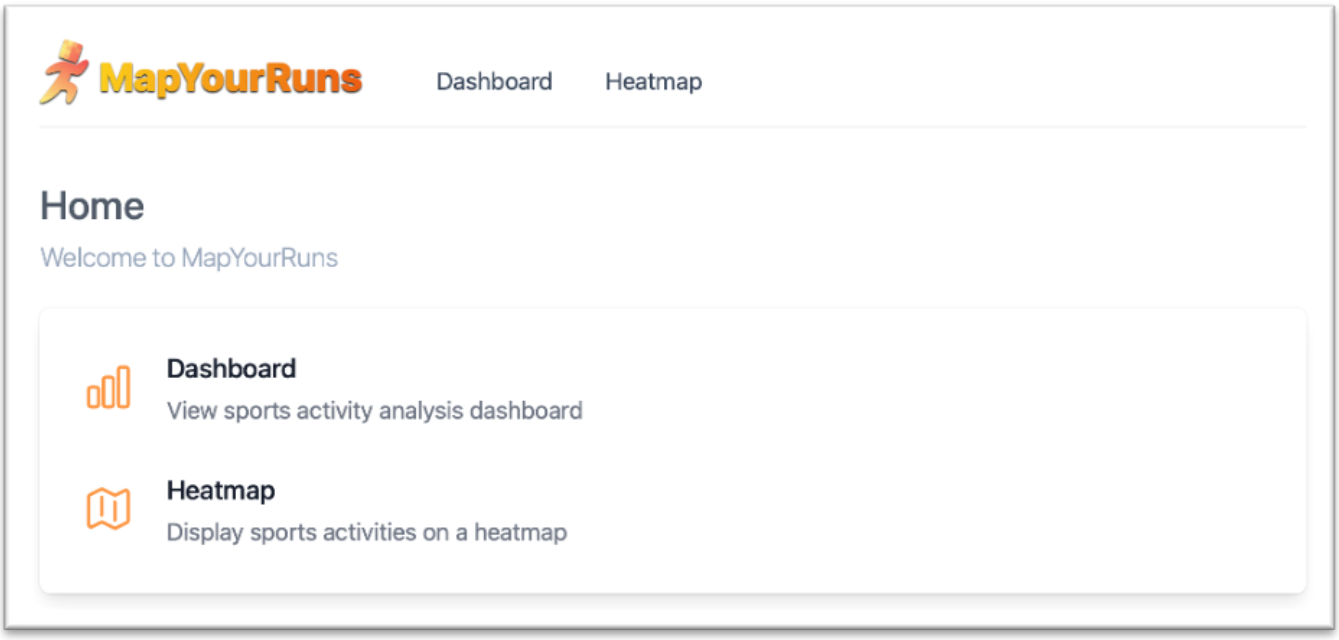
```
<div className="-mr-1 md:hidden">
    <MobileNavigation />
</div>
```

If switched to mobile navigation, a hamburger menu button appears, using pre-built components from the Headless UI library:

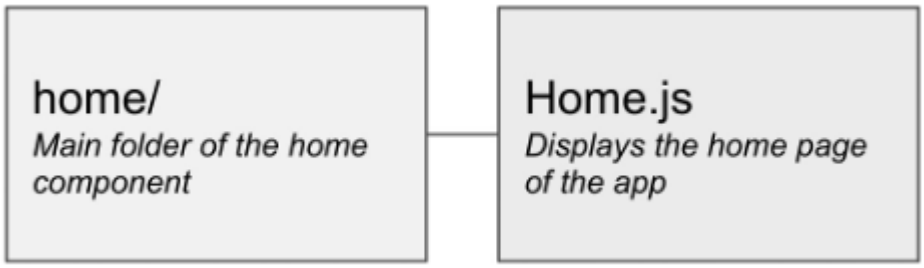
```
<Popover.Button
    className="relative z-10 flex h-8 w-8 items-center justify-center
[&:not(:focus-visible)]:focus:outline-none"
    aria-label="Toggle Navigation"
>
    {({ open }) => <MobileNavIcon open={open} />}
</Popover.Button>
```

4.3) Home Component

Component Preview:



Component Graph:



Description:

The file in the component `Home` inherits the code for the homepage of the web application. This serves as an overview of all sites the application has. The features are stored in a list with each list item being a dictionary that holds the necessary information to display the feature:

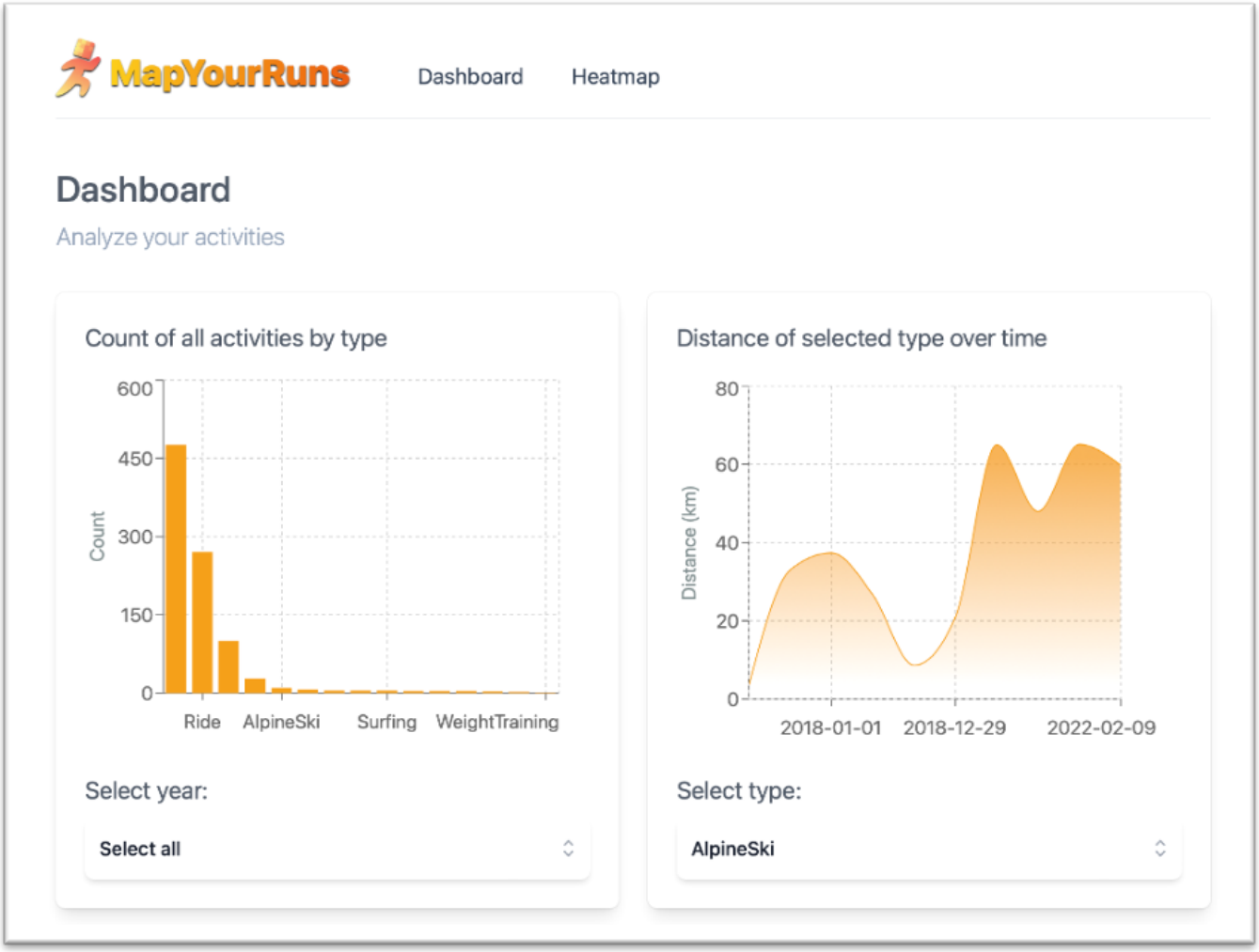
```
const features = [
    {
      name: "Dashboard",
      description: "View sports activity analysis dashboard",
      href: "/dashboard",
      icon: ChartBarIcon,
    },
    {
      name: "Heatmap",
      description: "Display sports activities on a heatmap",
      href: "/heatmap",
      icon: MapIcon,
    },
  ];
```

The features are then iterated over in the JavaScript-enabled HTML:

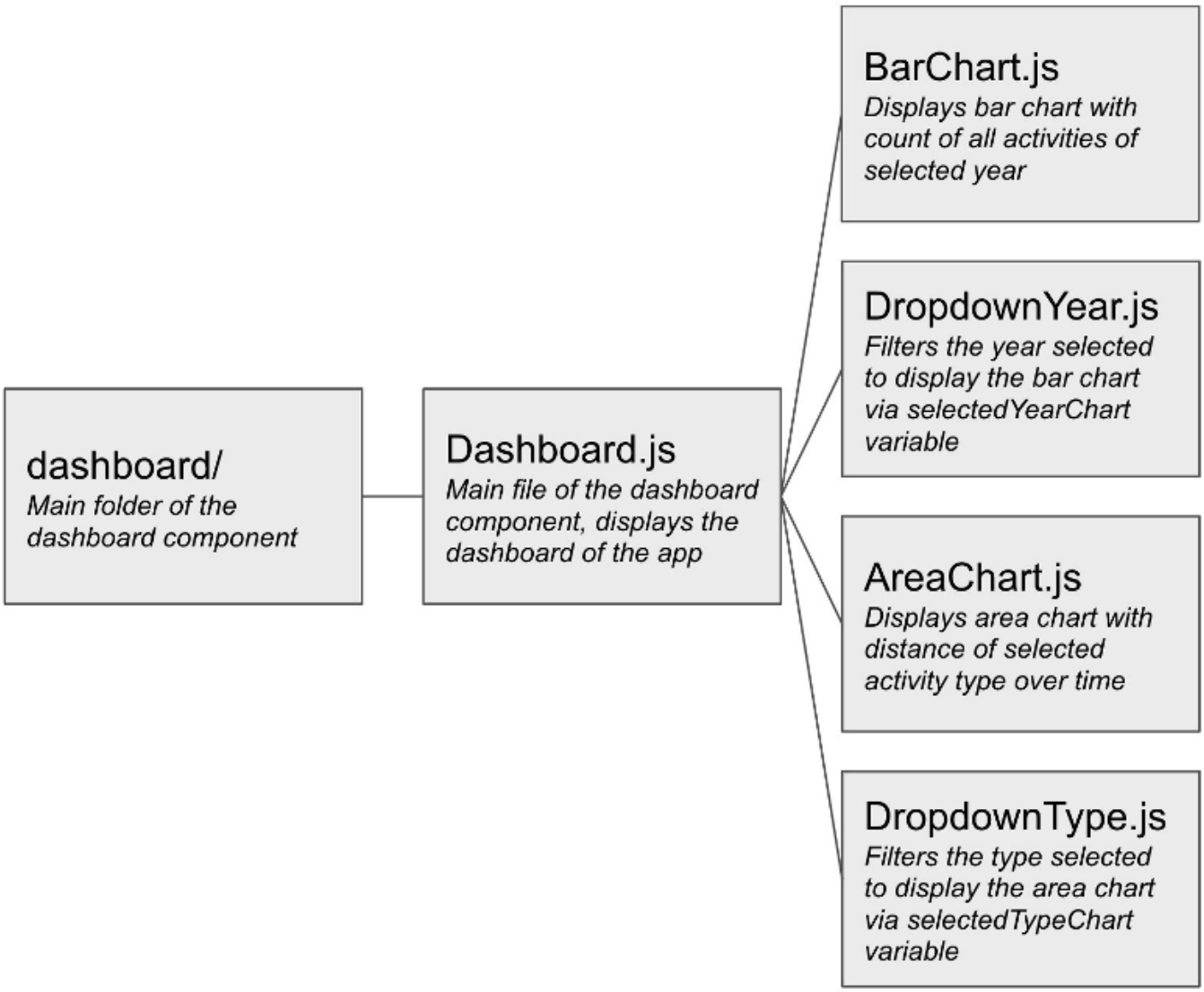
```
{features.map((item, index) => (
<div>
    <a
        key={item.name}
        href={item.href}
        className="-m-3 flex items-start rounded-lg p-3 hover:bg-gray-50"
    >
        <item.icon
            className="mt-2 mr-2 h-10 w-10 flex-shrink-0 text-orange-400"
            aria-hidden="true"
        />
        <div className="ml-4">
            <p className="text-lg md:text-xl font-medium text-gray-900">
                {item.name}
            </p>
            <p className="mt-1 text-md md:text-lg text-gray-500">
                {item.description}
            </p>
        </div>
    </a>
</div>
))}
```

4.4) Dashboard Component

Component Preview:



Component Graph:



Description:

The `Dashboard` component consists of a total of six files to not overload a single file with code.

Dashboard.js

The main file of this component is the `Dashboard.js` file which displays the dashboard of the web application. here, all the needed packages for the whole “dashboard” component are imported:

```
import React, { useState } from "react";
import { activities as activitiesFromFile } from "../../data/activities";
import { BarChart } from "./BarChart";
import { AreaChart } from "./AreaChart";
import { DropdownType } from "./DropdownType";
import { DropdownYear } from "./DropdownYear";
import { Container } from "../utils/Container";
```

The `Dashboard` component then stores all different activity types and activitiy years that are available in the data in a variable:

```
const allTypes = Array.from(
    new Set(
      activitiesFromFile.features.map((feature) =>
feature.properties.type)
    )
).sort();
```

```
const allYears = ["Select all"].concat(
    Array.from(
      new Set(
        activitiesFromFile.features.map((feature) =>
          feature.properties.start_date_local.substring(0, 4)
        )
      )
    ).sort()
);
```

The code then tracks the state of the dropdown and checks which activity the user wishes to filter for:

```
const [selectedTypeChart, setSelectedTypeChart] = useState(allTypes[0]);
```

```
const [selectedYearChart, setSelectedYearChart] = useState(allYears[0]);
```

With this information the actual data from Philipp gets filtered among the checked type of sports. Then the code visualizes the dashboard.

BarChart.js

The BarChart.js file displays a bar chart with the count of all activities of the specific year the user selected. First, we count how often each type of sports appears in the data set and stores this in a dictionary:

```
let typeCountDict = {};
  for (const item of activitiesFromFile.features) {
    if (selectedYearChart == "Select all") {
      if (typeCountDict[item.properties.type]) {
        typeCountDict[item.properties.type] += 1;
      } else {
        typeCountDict[item.properties.type] = 1;
      }
    } else {
      if (
        item.properties.start_date_local.substring(0, 4) ==
selectedYearChart
```

```
        ) {
          if (typeCountDict[item.properties.type]) {
            typeCountDict[item.properties.type] += 1;
          } else {
            typeCountDict[item.properties.type] = 1;
          }
        }
      }
  }
  typeCountDict = Object.entries(typeCountDict);
```

The dictionary then gets sorted by descending order. This means that the activities which the Philipp did the most, appear first:

```
  typeCountDict.sort(function (first, second) {
      if (first[1] < second[1]) return 1;
      if (first[1] > second[1]) return -1;
      return 0;
  });
```

Next the dictionary is converted into an array typeCountArray so that it is in the correct format to be consumed by the BarChartComp (from the Recharts library):

```
  const typeCountArr = [];
    typeCountDict.map((item, index) => {
      typeCountArr[index] = { type: item[0], count: item[1] };
  });
```

Lastly, the built-in React function `useEffect()` helps set up delayed chart loading for better responsiveness of the website, and finally, the `BarChartComp` displays the bar chart in the html of the return of the main function.

DropdownYear.js

The `DropdownYear.js` file filters for the year the user selected to display the bar chart via the `selectedYearChart` state variable, additionally using pre-built components from the Headless UI library. The code consists of JavaScript enabled HTML which we won't elaborate further here, as little JavaScript is used.

AreaChart.js

The `AreaChart.js` file displays the area chart with the distance of the selected activity type over the time. The code is similar to the `BarChart` component as it also stems from the Recharts library. The main JavaScript part is used to filter distances of activities where the `type` is equal to the `selectedTypeChart` as per the state variable:

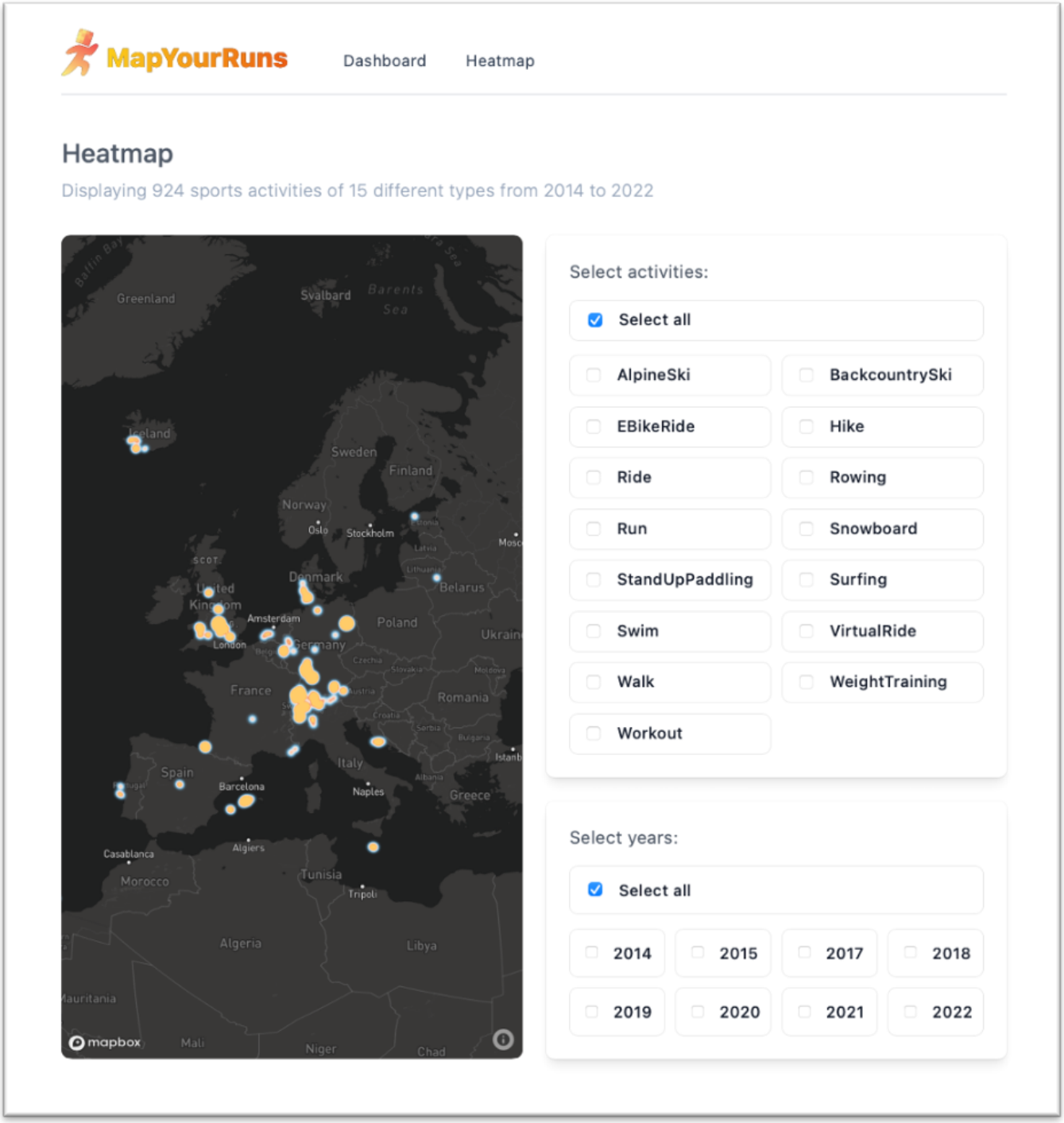
```
let typeLogArr = [];
  let counter = 0;
  activitiesFromFile.features.map((item) => {
    if (item.properties.type == selectedTypeChart) {
      typeLogArr[counter] = {
        type: selectedTypeChart,
        start_date_local: item.properties.start_date_local.substring(0,
10),
        distance: Math.round(item.properties.distance / 10) / 100,
      };
      counter++;
    }
});
typeLogArr.reverse();
```

DropdownType.js

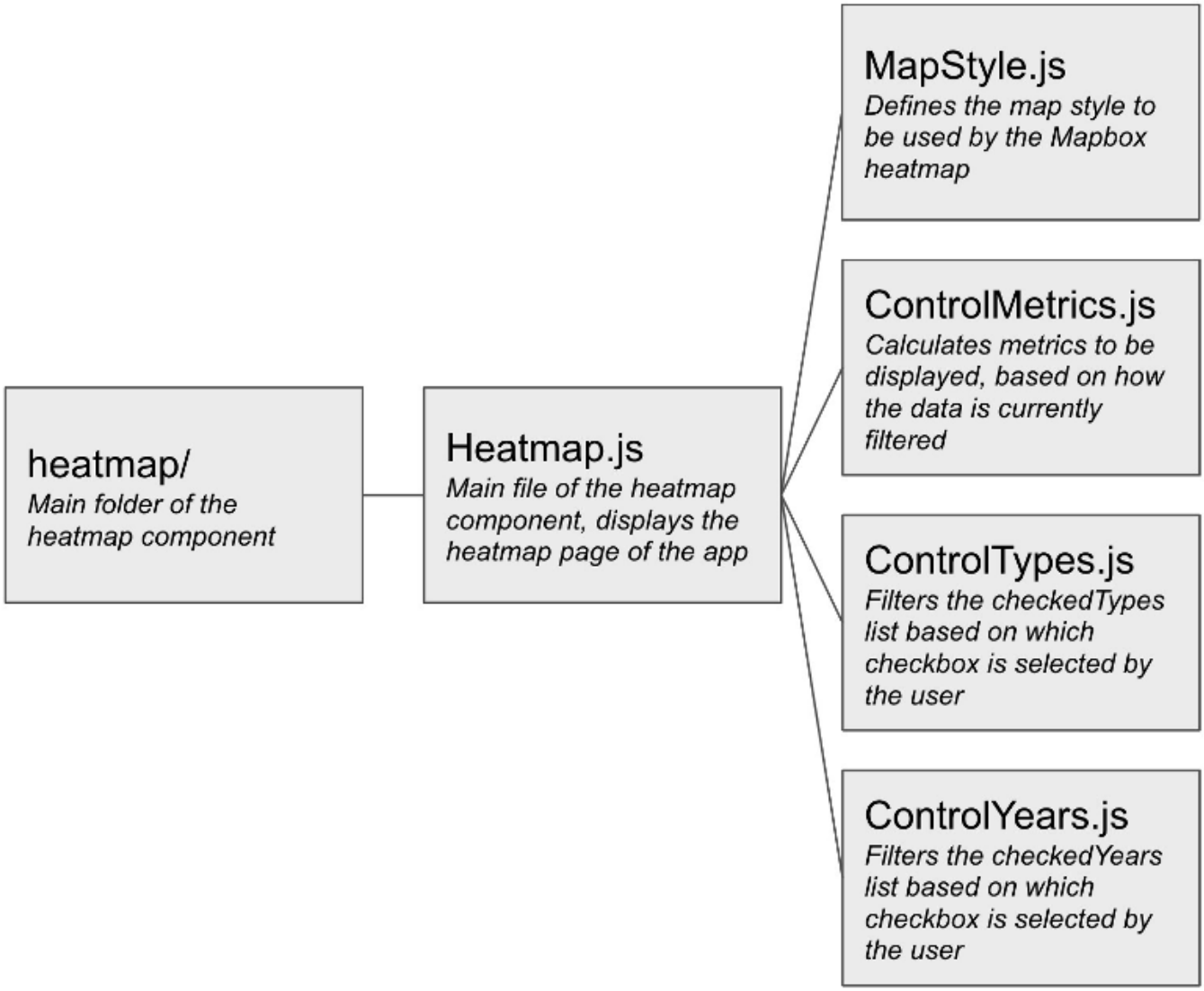
The DropdownType.js filters for the type of activity the user selected and displays the area chart we created before via the `selectedTypeChart` state variable. Similar to `DropdownYear.js`, the code consists of JavaScript enabled HTML which we won't elaborate further here, as little JavaScript is used.

4.5) Heatmap Component

Component Preview:



Component Graph:



Description:

The `Heatmap` component consist of a total of six files. The component is nested and therefore has different files to not overload a file with actions. This ensures a better functioning of the web application.

The main file of this component is the `Heatmap.js` file which displays the heatmap site of the web application. For the display of the heatmap, we use `MapGL`, a popular mapping component in React:

```
import MapGL, { Source, Layer } from "react-map-gl";
```

A token from Mapbox is needed to display the heatmap, which we have stored as an environment varibale in `.env.local`:

```
const MAPBOX_TOKEN = process.env.NEXT_PUBLIC_MAPBOX_ACCESS_TOKEN;
```

In the main `Heatmap` component manages the state varibales and passes them on to the other components.

The different types of sporting activities are stored in the variable `allTypes`, and the state tracked in `useAllTypes`, hence it is evaluated which filters the user has checked and therefore which data should be displayed in the map:

```
const allTypes = Array.from(
    new Set(
      activitiesFromFile.features.map((feature) =>
feature.properties.type)
    )
).sort();

const [useAllTypes, setUseAllTypes] = useState(true);

const [checkedTypes, setCheckedTypes] = useState([]);
```

The same is implemented for years:

```
const allYears = Array.from(
    new Set(
        activitiesFromFile.features.map((feature) =>
        feature.properties.start_date_local.substring(0, 4)
        )
    )
).sort();

const [useAllYears, setUseAllYears] = useState(true);

const [checkedYears, setCheckedYears] = useState([]);
```

With this information the actual data from Philipp's sports activities gets filtered among the checked options, and then re-converted into a vlaid `.geojson` object. Two functions are used to apply the filters to actually filter the `.geojson` file to be displayed by the heatmap:

```
const activitiesFilteredByType = activitiesFromFile.features.filter(
    (feature) => {
      if (useAllTypes == true) {
        return true;
      } else {
        return checkedTypes.includes(feature.properties.type);
      }
    }
);

const activitiesFiltredByTypeAndYear = activitiesFilteredByType.filter(
    (feature) => {
      if (useAllYears == true) {
        return true;
      } else {
```

```
        return checkedYears.includes(
          feature.properties.start_date_local.substring(0, 4)
        );
      }
    }
);

const activities = new Object();
activities.type = "FeatureCollection";
activities.features = activitiesFiltredByTypeAndYear;
```

MapStyle.js

The MapStyle.js file defines what map should be used to display the heatmap. It comes pre-defined from Mapbox, hence we won't detail it's functionality here.

ControlMetrics.js

This file calculates the number of activities that are currently filtered as well as the number of years that are filtered. It returns which types and years are filtered at the moment.

```
const metricNumActivities = activities.features.length;

const metricNumTypes = () => {
    if (metricNumActivities == 0) {
      return "0";
    }
    return useAllTypes == true ? allTypes.length : checkedTypes.length;
};

const metricYearFromTo = () => {
    let metricYearHelper = useAllYears == true ? allYears : checkedYears;
    metricYearHelper = metricNumActivities == 0 ? [] : metricYearHelper;
    const metricYearFrom =
      metricYearHelper.length == 0 ? "-" : metricYearHelper[0];
    const metricYearTo =
      metricYearHelper.length == 0
        ? "-"
        : metricYearHelper[metricYearHelper.length - 1];
    return metricYearFrom + " to " + metricYearTo;
};
```

In case that there is no data to the filters selected, the file returns a message indicating this. If there is data that matches the filter the code returns a message, which activities in which years are displayed right now:

```
const metrics = (numActivities, numTypes, yearFromTo) => {
    return numActivities == 0
      ? "There are no activities to be displayed based on your current
selection"
```

```
      : "Displaying " +
          numActivities +
          " sports activities of " +
          numTypes +
          " different types from " +
          yearFromTo;
};

const textColor = () => {
    return metricNumActivities == 0
      ? "text-red-400 text-lg md:text-xl font-light mb-2 mt-2 mb-8"
      : "text-slate-400 text-lg md:text-xl font-light mb-2 mt-2 mb-8";
};
```

ControlTypes.js

First, if a selectbox is clicked by the user from the HTML interface, that selection either adds or removes the checked item from the checkedTypes list:

```
const handleCheckType = (event) => {
    var updatedList = [...checkedTypes];
    if (event.target.checked) {
      if (useAllTypes == true) {
        setUseAllTypes(false);
      }
      updatedList = [...checkedTypes, event.target.value];
    } else {
      updatedList.splice(checkedTypes.indexOf(event.target.value), 1);
    }
    setCheckedTypes(updatedList.sort());
};
```

Another function is used to track if all types shoudl be displayed:

```
const handleCheckUseAllTypes = (event) => {
    if (event.target.checked) {
      setUseAllTypes(true);
      setCheckedTypes([]);
    } else {
      setUseAllTypes(false);
    }
};
```

Lastly, depending on which item is checked, we return a different Tailwind css class to be rendered in the HTML:

```
var isCheckedType = (item) => (checkedTypes.includes(item) ? true :
false);
```

In case there is no data to the filters selected, the file returns a message indicating this. If there is data that matches the filter the code returns a message, which activities in which years are displayed right now.

ControlYears.js

This file is built up like the ControlTypes.js file but checks for the different selected years instead of the different selected types.