Universitetet i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR THESIS

Study programme / specialisation: Computer Science	The spring semester, 2022 Open / ~~Confidential~~
Author: Stian Brekken Antonsen	Stian B. Antonsen (signature author)
Author: Erlend Bygdås	[illegible] (signature author)
Course coordinator: Erlend Tøssebro Supervisor(s): Magnus Særsten Book	
Thesis title: Visualizing COVID-19 data and search trends	
Credits (ECTS): 20	
Keywords: COVID-19 • Visualization • React • D3.js • Google Cloud Storage • BigQuery • Web application	Pages: 78 + appendix: 6 pages Stavanger, 15 May 2022

University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Visualizing COVID-19 data and search trends

Bachelor's Thesis in Computer Science
by

Stian B. Antonsen and
Erlend Bygdås

Internal Supervisors

Magnus Book

Faculty Supervisors

Erlend Tøssebro

May 15, 2022

“Visualizing is daydreaming with a purpose.”

Bo Bennett

Abstract

Since the COVID-19 pandemic's start, nations have published their COVID-19 data daily. There are already many sites to visualize this COVID-19 data. Many are tied to a specific nation, or will only give the user data for new confirmed cases and deceased. Few of the sites give an insight into how search trends changed during the pandemic. This project aims to show all of the data available to the user in relevant visualizations. The resulting application succeeds in giving clients more control over visualizations, although some of the tools did not get completed in the allotted time. The result is still a working web application that can give helpful insight.

Acknowledgements

We want to express our utmost gratitude to our supervisor Magnus Særsten Book for his excellent guidance through this project. His enthusiasm and technical knowledge gave us a quintessential push in the right direction, in addition to reading drafts and being available for questions seven days a week.

We would also take the time to thank our course coordinator Erlend Tøssebro for providing us with valuable feedback on our thesis drafts.

Lastly, we want to thank the Google Health team, who took the time to answer us, gave us helpful insight into their solution, and created the aggregated data set we used in this project.

Contents

Abbreviations

JSON	**J**ava**S**cript **O**bject **N**otation
DOM	**D**ocument **O**bject **M**odel
CORS	**C**ross **O**rigin **R**esource **S**haring
COD	**C**ovid 19 **O**pen **D**ata
UiS	**U**niversity 19 **O**f **S**tavanger

Chapter 1

Introduction

As of 2022, the world seems to be at the end of the COVID-19 pandemic. COVID-19 is a respiratory disease caused by severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2) [1]. For most healthy individuals, the disease only causes mild or moderate symptoms. Even if most individuals were fine, the problem of the disease was the sheer number of people who would need medical assistance as the disease spread rapidly.

The earliest known case in a human was on the 1st of December, 2020 [2]. However, the first time the World Health Organisation (WHO) got information about a confirmed case of SARS-CoV-2 was on the 31st of December, 2020, in Wuhan, China. A month later, WHO declared a "Public Health Emergency of International Concern" on the 30th of January, 2020. The COVID-19 pandemic was officially declared a pandemic by WHO on the 11th of March, 2020. The last couple of years has seen many changes to ordinary life. At the tail end of the pandemic, we should look back on the data accumulated and hopefully learn from this data. This thesis aims to create an application for visualizing publicly available COVID-19 data.

Understanding how our attitude to COVID-19 changed during the pandemic is something many visualizations do not show. An indication of people's attitudes can be extracted by studying people's search trends and mobility. Our application is a working web application with different ways to display data related to COVID-19. The challenge was to create a web application that could show detailed data for individual nations and give a complete overview of the global situation. Therefore, it was vital to create visualizations that could single out regions and allow the possibility to compare and give a user an overview of a larger part of the data set.

1.1 Motivation

We felt personally motivated to learn more about the pandemic that impacted our social life and education. One primary motivator was to lower the threshold for gaining insight into the data accumulated over the last years. The application should create a simple and accessible way to view the data so that anyone can access it.

Another reason was that we could emphasize different parts of the data compared to other sites. Allowing the user not to be restricted to only premade graphs and be allowed to create a visualization with any part of the data.

1.2 Problem Definition

The COVID-19 pandemic has been widely studied and analyzed since it started in 2020. Most of the visualizations and analyses focus on the number of cases and deceased. Having interactive visualization showing how search trends have changed during the pandemic could give insight into how they are affected by other data.

For this project, the goal is to create a website that allows users to explore COVID-19 data. Users can view how search trends, confirmed cases, deceased, along with other COVID-related data such as hospitalizations and vaccinations have changed since the start.

It should be possible to navigate through the data on a daily basis and compare daily trends. Additional data could also be incorporated to give a complete sense of the various changes over time.

1.3 Source Code

The source code for the project is available on GitHub `https://github.com/bachelor-group/bachelor-group.github.io`, and the web application is hosted at `www.bachelor-group.github.io`.

See Appendix A for instructions on how to run the web application locally.

1.4 Outline

Chapter 2 provides the essential background theory to properly understand the solution. The chapter includes information about the frameworks used and an introduction to some of the concepts used when visualizing data on a map.

Chapter 3 describes the end product with an overview and a more in-depth explanation of essential components in the application.

Chapter 4 discusses the solution from idea to finished product. It gives the reader an insight into some of the challenges with certain approaches. Earlier approaches are also discussed.

Chapter 5 concludes the project by giving a summary of the project and gained knowledge, as well as describing future directions.

Chapter 2

Background

This chapter describes the different technologies used in the development of the application. Technologies such as React, TypeScript, D3.js, Natural Earth, and Geo-/TopoJSON will be explained. The COVID-19 data set provided by Google will also be described. Why some of these technologies were chosen over others is described in chapter 4.

2.1 Theory

This section describes fundamental concepts essential to understanding the project.

2.1.1 Maps

One of the plethora of ways to visualize data is using maps. This section will explain the terminology used in this thesis. The maps used in the project use different administrative levels, often just called **admin-level**. The admin-level refers to the administrative level used for geographical features shown on the map. The admin levels are in a hierarchical system where admin-level zero is the highest, and the next level (i.e., level one) is the level beneath, and so on. This project supports the three highest levels, as these are the levels provided by Google's open Data repository (COD). Table 2.1 has an overview of the levels available.

Table 2.1: Table showing the different administrative levels used in this project.

Admin Level	Description
0	The highest government level. This is where we find nations, e.g. United States, Norway, ...
1	The second-highest government level. Meaning the largest sub-national unit. The name will differ depending on the nation, but examples are states (USA), counties (Norway), prefectures (Japan), ...
2	The subdivision of Admin level 1. The name will differ depending on the nation, but the data-set used only support data from the United States at this level. Meaning this would be referred to as counties

Throughout this thesis, the admin level will be used to describe the maps instead of names such as states, counties, or other similar terminology, to reduce confusion.

It is worth noting that there are several different types of maps that can be used to display data. This project arguably uses the most common type called a choropleth map [3]. Choropleth maps are thematic maps that link geographical units to data. The data set that was used linked its data to geographical regions, making a choropleth map an excellent fit for this project. Choropleth maps are best used with continuous and normalized data. If not, some regions would naturally be misrepresented due to a high/low population. Every time a choropleth map is used in this project, the map is normalized by population, and usually to data per 100 000.

Figure 2.1 shows the equirectangular projection, which was chosen for this project, this decision is discussed in Section 4.6.2.

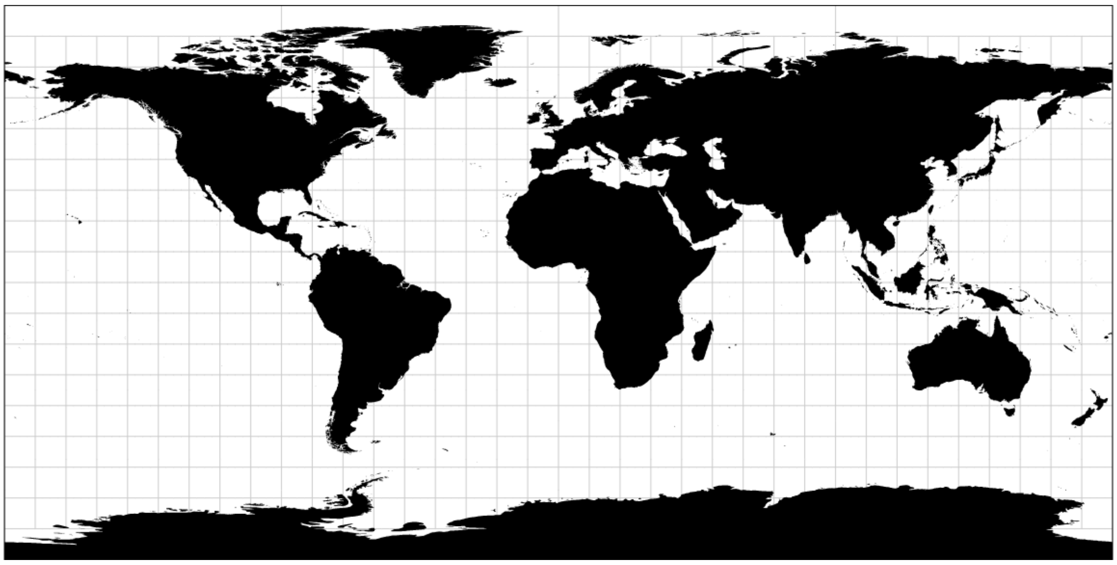


Figure 2.1: How the equirectangular projection looks. Image is from [4].

2.1.2 Location Codes

Multiple different types of location codes are used and referenced in the project. ISO31662, NUTS, FIPS, and post codes are all used. ISO codes are internationally recognized codes that represent every country and most dependent areas [5]. The codes are found in both two and three letters, Alpha-2 and Alpha-3, respectively. Alpha-2 is used in top-level domains on the internet, such as `.no` for Norway and `.se` for Sweden. Alpha-3 is often used in international sporting events, e.g., NOR for Norway, and SWE for Sweden. The code `NO_11` is the ISO 3166-2 code for Rogaland, Norway.

Nomenclature of Territorial Units for Statistics (NUTS) is a standard for referencing subregions for countries, often in an statistical context [6]. NUTS codes for certain subregions can differ from the official administrative divisors in that country. NUTS begins with a two letter code representing the country, and for each deeper subregion level an extra number is added.

Federal Information Processing Standard Publication 6-4 (FIPS) are only used in the US to describe counties

2.2 Code Libraries and Frameworks

The different libraries and frameworks used will be explained. Reasoning behind these choices are presented in Section 4.2.

2.2.1 TypeScript

Since JavaScript's release in 1995, it has been the standard in client-side scripting for web browsers [7]. TypeScript is a superset of JavaScript, which means it offers all the same features, but with type checking [8]. TypeScript code is converted to JavaScript, meaning it can run wherever JavaScript can. JavaScript is an interpreted language that is only interpreted at runtime, which means there is not much feedback when writing code. Since TypeScript is a typed language, it helps intelligent code completion provide more feedback to developers, preventing potential crashes or bugs. It is also an object-oriented programming language [9], which is something students at UiS are familiar with. Typed languages increases the readability of the code, and this is because it is easier to understand the true intention of the code when types are present. It also provides information that would otherwise be commented in the source code. TypeScript was used as the frontend scripting language.

2.2.2 React

The most used frontend framework in 2021 was React [10]. React allows for the creation of components that can easily be reused. Manipulating the DOM is done very efficiently in React by using something called the Virtual DOM. The Virtual DOM is a local copy of the HTML DOM. React updates its Virtual DOM before it syncs the changes to the real DOM. This way only differences are synchronized instead of refreshing the whole DOM, which would result in slower performance. This is called Reconciliation [11].

React uses a Diffing Algorithm to update its Virtual DOM before syncing with the HTML DOM. The algorithm compares the Virtual DOM before and after changes, checking only for differences. An example is the two following HTML structures.

```html
<div>
    <MapComponent />
</div>

 <!-- and -->

<span>
    <MapComponent />
</span>
```

Listing 2.1: HTML structures using different tags.

Since the tags `div` and `span` are different, React will update the whole Virtual DOM. This forces the component `MapComponent` to be remounted as well. When tags are equal, React will look at its attributes to decide if they are equal or not. Two div elements with different id's will be considered different. For any equal tag elements, React also checks for child elements. React requires child elements of a list to contain a key property, in order to check for equality in elements. Consider the list in Listing 2.2.

```html
<!-- before -->
<ul>
    <li>child1</li>
    <li>child2</li>
</ul>

<!-- after -->
<ul>
    <li>child1</li>
    <li>child2</li>
    <li>new child3</li>
</ul>
```

Listing 2.2: Adding an extra child element to a list.

React iterates through each list, and matches each child element. Because the first two are equal, only the latest addition, i.e., the new child in list two, is added to DOM. Consider another example where a new child is added to the beginning instead of appending an element to the end of the list. As React iterates over the list, every element will be different because elements in list two have changed indices. By requiring key properties to each child element, react will know if elements are new or not. The keys stop React from rebuilding the entire list repeatedly, leading to better DOM manipulation performance.

React will not be interactive by itself, it has to be told when there is an update. Some frameworks, such as VueJS, use a `watch` which keeps track of updates of a state. When using React hooks, the component will re-render whenever the setter function for the hook is called. For example the `useState` hook:

```typescript
const [data, setData] = useState<string>("initial data");

function changeState() {
    // setter
    setData("new data");
}
```

Listing 2.3: Example of the react hook useState, and using its setter.

When the function on line three in Listing 2.3 is called, react will update its state, and the component will re-render.

Components in React help with splitting the UI into independent, reusable pieces, and allows developers to think about each piece in isolation [12]. There are multiple ways to define components in React, and one is to declare a function that takes properties (often referred to as props) as arguments. The functional component will then return a JSX element. A JSX element is a syntax extension to JavaScript [13]. JSX elements combines JavaScript with HTML, which means JSX comes with the full power of JavaScript [13].

State management is a term that combines how to store state and how to change it [14]. It is essential in every web application React offer many different ways of storing state. Variables can be kept locally in a component by using the react hook `useState` or `useReducer`. Another option is keeping data in a store by using one of many available third-party libraries, e.g., Redux. Also, keeping data on the `window` object globally is a choice, although it is not recommended, especially for bigger projects.

2.2.3 D3.js v6

Data-Driven-Documents [15] is one of the most used charting frameworks for JavaScript [16]. It also has excellent support for types, and can easily be used with TypeScript. D3.js simplifies the process of making graphs and charts. Every graph, map, and other visualizations in this project was created with the help of D3.js. D3.js works by binding data to the DOM, then applying the transformations to the document [15]. It efficiently manipulates documents based on received data. This approach avoids proprietary representation, provides excellent flexibility, and uses the full capabilities of HTML, SVG, and CSS [15]. D3.js can use HTML elements, SVGs, and Canvas to create visualizations. SVG is explained more in-depth in Section 2.2.4. This leads to minimal overhead and support of large data sets, as well as dynamic behaviors for interaction and animation [15].

Modern browsers support D3.js selectors. Elements can be selected using D3.js's `d3.select (<CSS Selector>)` and `d3.selectAll(<CSS Selector>)`. After an HTML element is selected, the style and all other attributes can be changed. This can be done using the `attr()` method. D3.js selection is simply an array of nodes [17]. D3.js' programming paradigm is similar to functional programming and it is common to use many method's at once, such as `selection.function().function().function()...` The selectors are defined by W3C Selectors API [18].

The D3.js graph gallery [19] displays a collection of all available chart types. There are many charts with simple examples, and the source code is available, so they can easily be reproduced.

2.2.4 Scalable Vector Graphics (SVG)

SVG is a vector image format for graphics, supporting interactivity and animations. SVG is an open standard by World Wide Web Consortium (W3C) [20]. W3C develops protocols and standards for the World Wide Web. They are defined in the XML format, and it integrates well with other W3C standards, such as the DOM. Since SVGs are vector graphics, the quality of SVG images remains the same as the vector is scaled up or down. This works great when there is zooming involved, because there is no loss of quality.

2.2.5 Google Cloud Platform

Google Cloud Platform is a suite of cloud computing services that runs on the same infrastructure that Google uses internally for its products [21]. BigQuery by Google is a serverless, highly scalable, and cost-effective multi-cloud data warehouse designed for business agility [22].

Google Cloud Storage allows customers to store any data, no matter the size. Data can then be retrieved at any time, as many times as a user wants. Data is stored in buckets. Buckets are the containers that hold data. Everything in the Cloud must be stored in buckets [23].

2.2.6 Natural Earth

Natural Earth is a public domain map data set that allows for the easy creation of visually pleasing and well-crafted maps [24]. Natural Earth is one of many Geographical Information Systems (GIS) that are used to model the real world into convenient data models, which can be used to create maps. GISs have two main types of data; raster images and vector data. Natural Earth contains both of these types. For this thesis, the primary use case of Natural Earth was to create a thematic map. Therefore, it was only necessary to use the shapefiles that were in the ArcGIS format containing vector data. Natural Earth uses a consistent way of drawing its vectors, making it possible to use different administrative levels in the same visualization. Natural Earth's data is available at multiple scales, 1:10 million, 1:50m, and 1:110m. All of this means a visualization can dynamically change the level of detail, for example, depending on the zoom level.

In addition to getting geographical data, Natural Earth also offers non-spatial data, which is data that does not refer to geographical location. This data includes crucial information such as location codes following global standards, populations, and names written in several languages. Having standardized codes creates a convenient way to connect data to each feature. This data is stored in a shapefile; a generic term used to denote a directory containing at least these files `.shp`, `.shx`, and `.dbf`. The `.shp` file contains the actual feature geometry. `.shx` is an index file containing indexes to the geometry. Lastly, the `.dbf` file is a database file that contains attributes associated with each geometry object. This file structure means there is a one-to-one relationship between geometry and attributes [25]. It may also have other optional files with even more information, but they are not relevant to this project. The strict guidelines for shapefiles help convert the file to other relevant file formats like TopoJSON, GeoJSON.

2.2.7 GeoJSON

GeoJSON is an open standard of JSON (JavaScript Object Notation). JSON is a lightweight format for exchanging data. The power of JSON comes from it being easy to read or write for both humans and computers [26]. JSON also works great together with JavaScript since they use the same structure for objects. GeoJSON is a proper subset of JSON, meaning that GeoJSON follows all the rules of the JSON standard, but also has different specifications to make it GeoJSON. While GeoJSON encodes the same data as a shapefile, GeoJSON does this in only *one* file. GeoJSON is not a convenient option for maintaining data, as both spatial and non-spatial data are encoded into one singular file. There is no separation of concern, which makes modifying the data more complicated than necessary. However, for a consumer of ArcGIS shapefiles (for example, this project), having to index between spatial and non-spatial data is possible, but cumbersome. Because GeoJSON combines the data, it is more efficient than its shapefile counterpart, as it is not necessary to index in this combined format. Instead of going into excruciating detail in plain text about how GeoJSON is structured, please refer to Listing 2.4 beneath to get a sense of it.

```json
{
  "type": "FeatureCollection",
  "features": [{
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [102.0, 0.5]
      },
      "properties": {
        "prop0": "value0"
      }
    }, {
      "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]
        ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": 0.0
      }
    }, {
      "type": "Feature",
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [
            [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],
            [100.0, 1.0], [100.0, 0.0]
          ]
        ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": { "this": "that" }
      }
    }]
}
```

Listing 2.4: Example from the GeoJSON standard [27]. Note some formatting has been done to reduce the length of the original example.

From Listing 2.4 it can be seen that each geometric object is called a feature. Each feature will have a type, e.g., Point, LineString, Polygon, MultiPoint, MultiLineString, or MultiPolygon. Every feature will also have non-spatial attributes in the key called properties. The coordinates are the spatial data and are denoted in decimal degrees of longitude and latitude. Because of this, GeoJSON data is independent of any projection, and a user may choose a projection that they deem fit for the project.

2.2.8 TopoJSON

GeoJSON has a list of features where all of the data associated with one geometry can be retrieved at one index. This structure makes the format great to work with. However, for storing, GeoJSON can be improved. Each feature has coordinates describing its borders, which means that any shared border between features is duplicated, creating much redundancy. As an example, take USA and Canada. Their coasts are unique, and there is no way to remove any data without losing information, but both USA and Canada will also store their shared borders. Instead of storing the same data twice for each nation, one could store it once and have each nation reference the stored data.

TopoJSON extends GeoJSON and removes this redundancy of GeoJSON. In a TopoJSON file, topology is stored instead of discrete feature objects. Instead of each country storing the same border, they will store a reference to a line/arc. So shared borders are now only defined once, instead of multiple times, thus reducing file sizes [28]. There is also the benefit of shape simplification with TopoJSON, where file size can be reduced further at the cost of some spatial information. Since this project only uses TopoJSON for storing files, this thesis will not detail how the format is structured or how TopoJSON works. For further reading, the creator of TopoJSON, Mike Bostock, created a blog post that explains TopoJSON in further detail [28].

2.2.9 MapShaper

MapShaper is a tool for editing spatial files such as shapefile, GeoJSON, and TopoJSON. There are different ways of using the tool. MapShaper is available as a command line tool and it can be run locally in a browser. There is also a hosted version on `MapShaper.org`. It can convert files between different formats and reduce the overall file sizes.

2.3 COVID-19 Open Data

Google has one of the most comprehensive data repositories for COVID-19-related data, called the "COVID-19 Open Data Repository" (COD). The data repository is an aggregation of data from several well-trusted sources worldwide [29]. With the use of Google's data repository, one could gain insight from data worldwide in the same way. Google divides its sources into three categories; authoritative sources (governmental, health, universities), general sources (news media, publications), and crowdsourcing (volunteers, contributors). Google lists, as of writing, 302 different sources for their data repository [29].

The repository provides data in both JSON and CSV format. There are also different files for all data categories. For example, one can use the epidemiology file to get data only relevant to epidemiology. This structure creates a simple way for users to ask for files of smaller sizes. The different categories available are: Aggregated, Index, Demographics, Economy, Epidemiology, Emergency Declarations, Geography, Health, Hospitalizations, Mobility, Search Trends, Vaccination Access, Vaccination Search, Vaccinations, Government Response, Weather, WorldBank, By Age and By Sex. Google also provides the ability to get the latest data from their API by adding `/latest/` to the URL before the file. This request will return the latest data row for each location. In addition, the Google API allows for querying specific locations to get an aggregated file for said location. This is done by using `location/`⟨*location_key*⟩`.csv`. Locations keys are built using ISO-3166, NUTS, FIPS, and postal codes. These are standardized codes to make it easier to use with any library that also uses these standardized codes. The data is structured to use location keys and the date to identify rows in the file uniquely.

Chapter 3

Solution Approach

In order to understand the application, a thorough understanding of the project structure and how core components work is imperative. This chapter presents the final state of the web application. Some early ideas are also briefly discussed, but these are discussed more in-depth in the next chapter.

3.1 Project Structure

The project structure was one of the earliest things decided upon. As React was the chosen framework for the application, the web pages would be split up into React components. An overview of the file structure for the repository is shown in Figure 3.1.

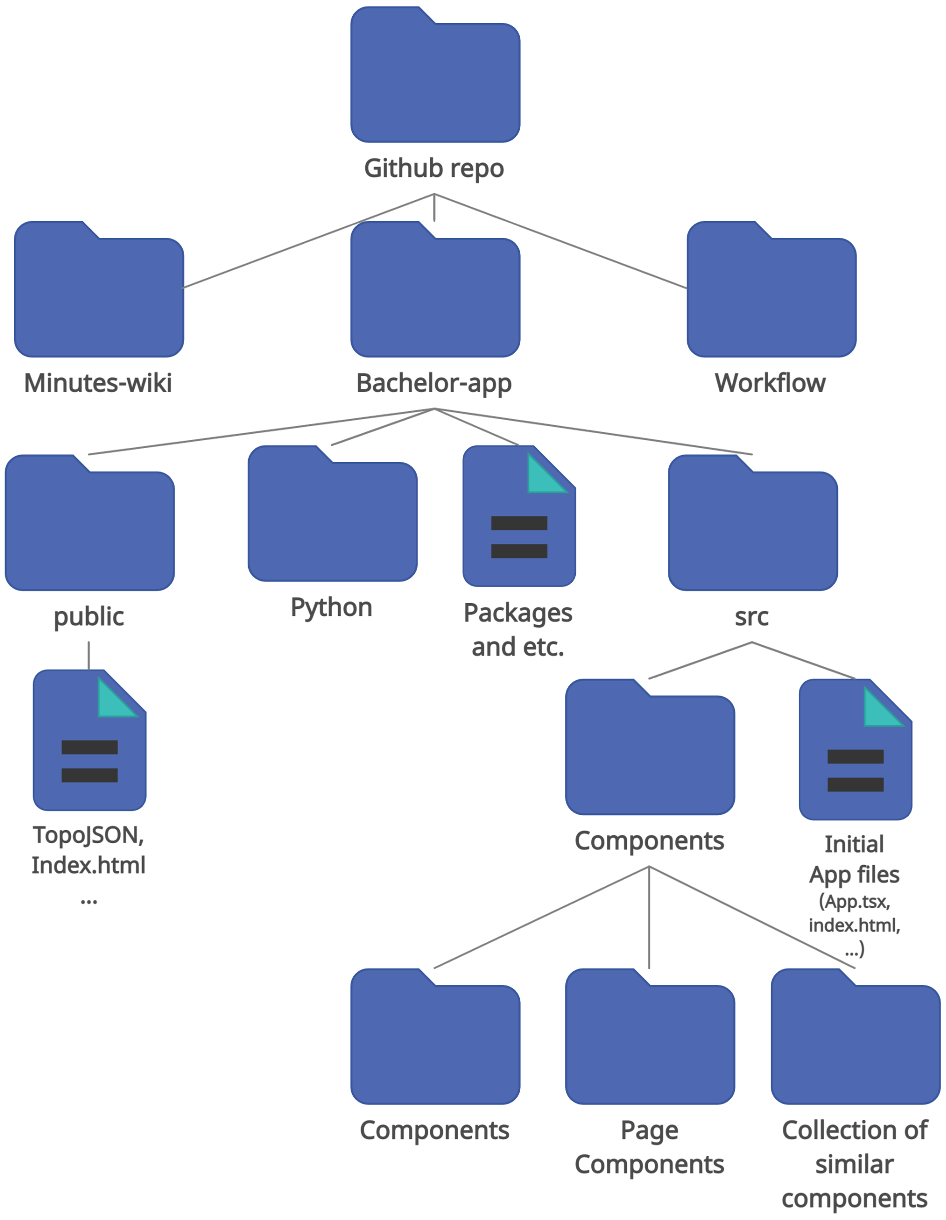


Figure 3.1: Overview of file structure for the application.

This file structure was sufficient for this project, as it had a separation of concerns. For example, the folders that are marked as compound components are components complex enough that they need to have sub-components. An example of a compound component is the `Map` component, which has a subcomponent called `DrawMap`. Page component folders contain the main component for a page and any unique sub-component not needed for any other page. The collection of similar components is a folder with components that serve a similar purpose. For example, the project's graphs folder contains all components that are charts or plots for graphing COVID-19 data.

The public folder includes all files that are resources publicly available from this application. This folder is where the home page `index.html` is served to a client. It also includes the TopoJSON files created for this application. These files were created using Natural Earth's cultural maps, which were initially ArcGIS shapefiles. The files were then loaded into the Mapshaper tool hosted at `MapShaper.org`. The file sizes were then reduced as much as possible while keeping an acceptable level of detail in the maps. Finally, the files were converted into the TopoJSON format. A Python script which was created for this project, then added a location key column in the TopoJSON file. Once the `LOCATIONKEY` column was added, the TopoJSON files were uploaded to the GitHub repository. The TopoJSON files in this project do not have all the raw data from Natural Earth but are reduced by about 50%, by reducing the resolution of the vector data. It should be noted that the client always converts the TopoJSON files into GeoJSON when the files are used in the application. GeoJSON is the preferred file format in D3.js, and most functions support this format.

3.2 Pages

There were no specifics in the project description of what pages the web application should contain. The structure of the pages to the web application can be summarized in the visualization in Figure 3.2.

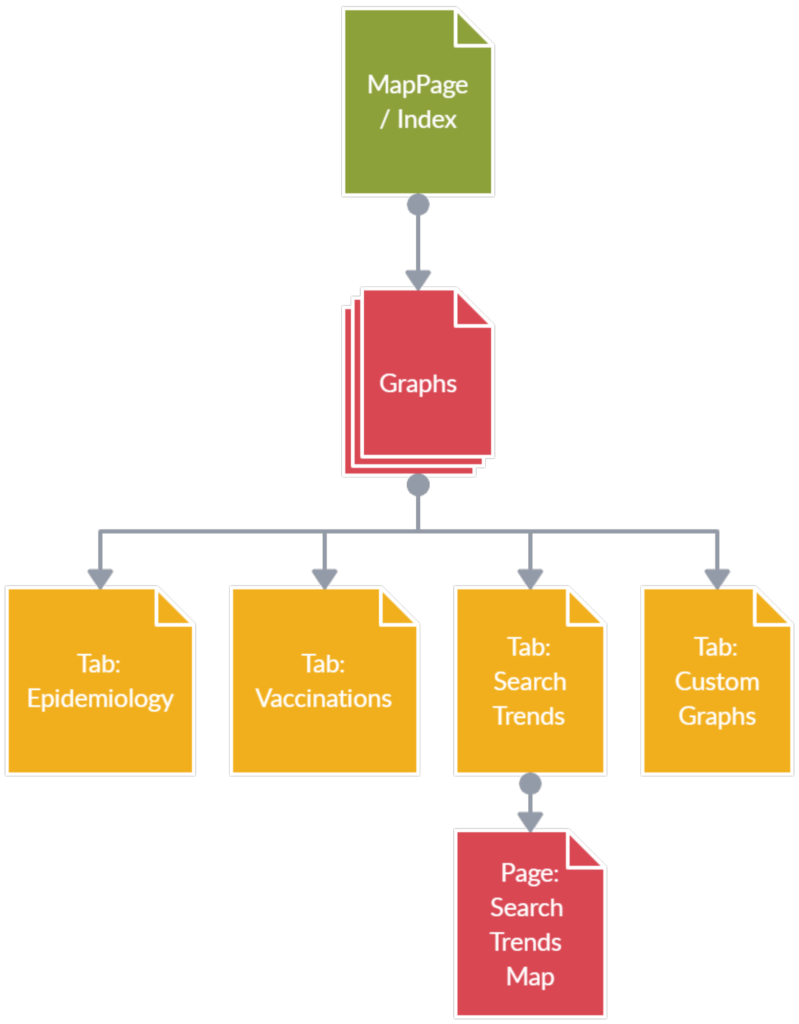


Figure 3.2: The page layout of the web application.

The index/MapPage is the page served to the user when entering the site. A screenshot for this page can be seen in Figure 3.3. The graph page is a page that contains several tabs within it. The tabs are shown in Figure 3.2 as the yellow pages. A screenshot of how the graph page looks with the custom graphs open is provided in Figure 3.6. Lastly, a separate page is accessible with a map of search trends. However, this page is only available for a few countries that have search data available. An example of this page is shown in Figure 3.8.

3.2.1 Front Page

The front page consists of multiple components: `Sidebar`, `MapComponent`, `DateHistogram`, and the animation component called `Animator`. Figure 3.3 shows a screenshot of the final design of the front page.

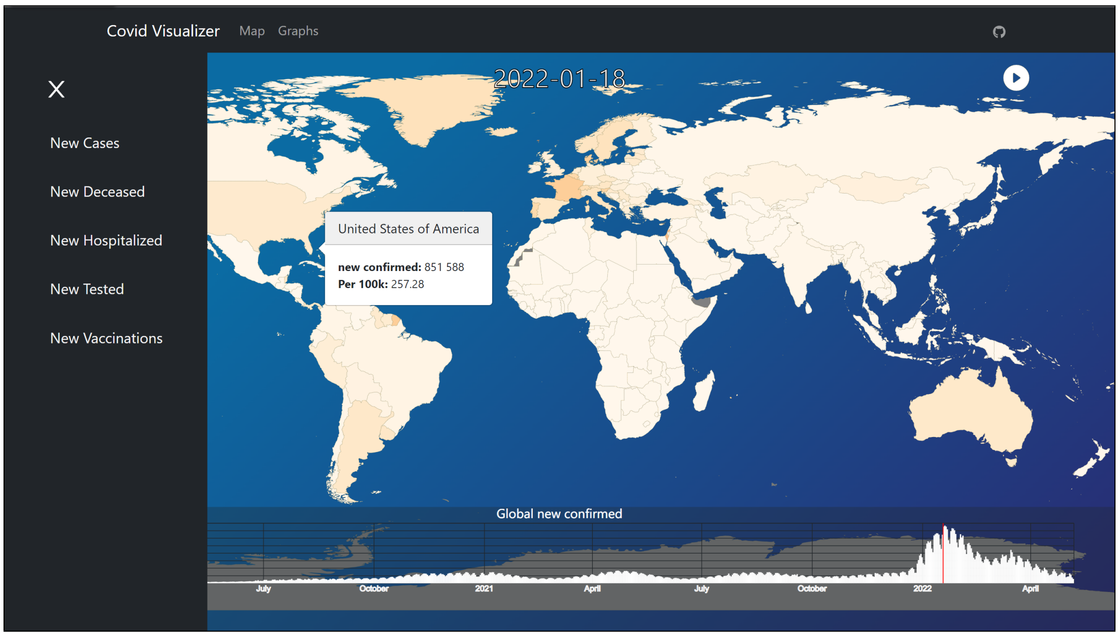


Figure 3.3: Design of the front page for the web application.

The color scale used in the map uses an interpolator between orange and red, called `interpolateOrRd` from the d3-scale-chromatic package [30]. The colors available in the interpolator can be seen in Figure 3.4. The color scale uses the D3.js scale sequential function and the interpolator to create a function that can turn data values into the corresponding color. The domain chosen is 0 to the maximum value throughout the entire pandemic. Nations with a population of less than 1 000 000 are not used when finding the maximum value for the domain to reduce the impact of outlier values from smaller nations. The reasoning behind this is discussed further in Section 4.6.3.

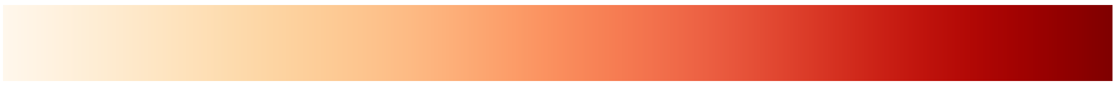


Figure 3.4: The gradient of colors used to map data to colors.

In cases of missing data, the app also uses grey and magenta based on why the data is missing for a country. If a country does not publish data for the selected data property, there is no column in the CSV file for this data, or the data points may either be null or an empty string. When the map can find data for a geographical feature but cannot find the currently selected column, the location is displayed in gray. There are also situations where the app was unsuccessful in finding data related to a geographical feature from the GeoJSON. This may be because the COVID data does not exist for this location. The location will then be displayed as the color magenta to mark it as an error. Therefore, the user should not expect the region's color to change when the date is changed. The situations where this may happen will be further discussed in Section 4.5.5, but can frequently be seen with administrative levels higher than zero.

The sidebar, which is expanded in Figure 3.3, allows the user to choose a filter for which data to display, such as new cases, new deceased, and new hospitalization. The default value of the sidebar is new confirmed cases. Seven days from the current date is the default date shown when opening the page. This ensures that all countries have had time to update their data. Section 4.5.1 explains in more detail why a lag of seven days was chosen.

When hovering over a region, a tooltip appears with the exact number of cases and cases per 100k population for whatever data type is selected. The user can also click any region displayed on the map, which will have different effects depending on the admin level. When a location is clicked, the application loads all subregions for that location. An example of this functionality is shown in Figure 3.5. The tooltip will then show the name and data for the chosen subregion instead of the location. A user can zoom in on the map using the mouse wheel, or with a pinching motion on a touchpad or touch screen to see the smaller geospatial features.

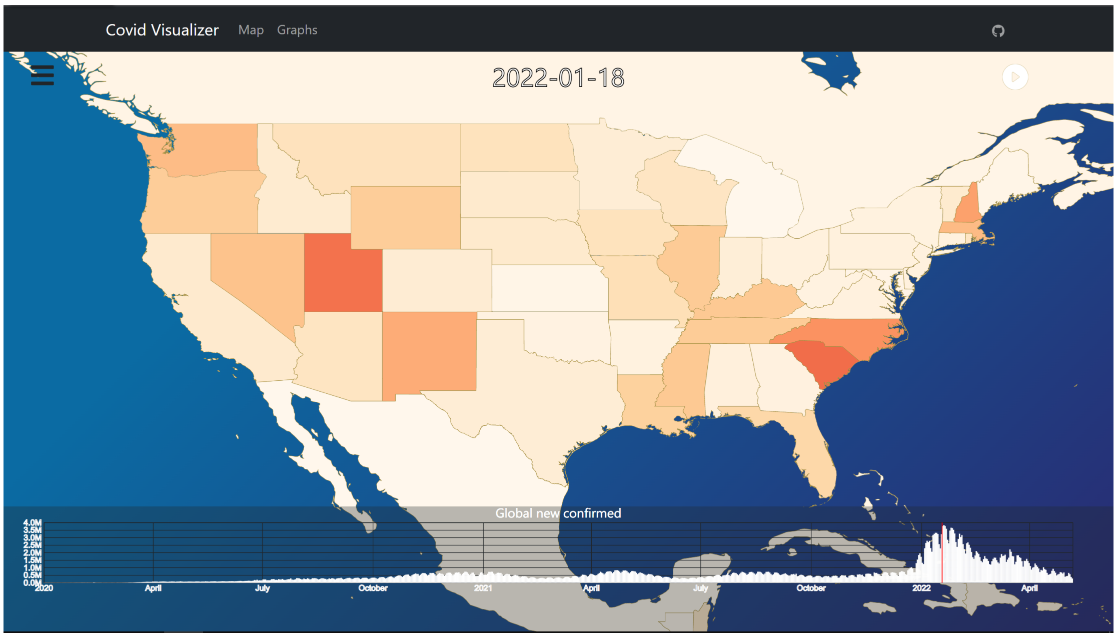


Figure 3.5: When a user clicks on a country the subregions for that country is displayed.

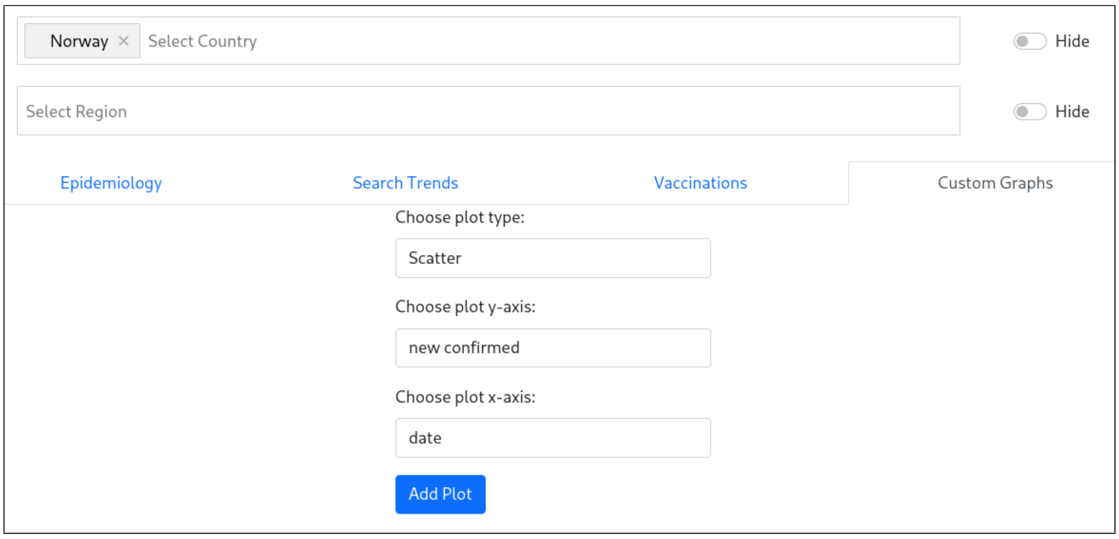
The histogram at the bottom of the page displays an overview of selected data temporally. Users can click on the histogram to change the data displayed to the chosen date.

The play button in the top right on Figure 3.3 is from the animator component. It starts an animation from the currently selected date until the last day in the data set when it is clicked.

3.2.2 Graphs Page and Navbar

All graphs are shown on a single page with tabs for some categories of the COVID-19 data. By having all graphs on the same page, the app can keep the state when switching between tabs, meaning there is no need to reload any data. This approach primarily works because the files retrieved by this page are the aggregated data for each of the selected locations. Since the aggregated data has many data categories, the app loads the data for all tabs. Thus, changing the tab would only need to change what graphs are displayed without loading additional data.

This approach means an adequate initial load time and an instant load when changing between tabs. To give the user more flexibility, it was decided early on to give users a separate tab where they can choose any data they want in any of the implemented graph types. Figure 3.6 shows the graph page with the four different tabs: epidemiology, search trends, vaccinations, and custom graphs.



Choose plot type: Scatter

Choose plot y-axis: new confirmed

Choose plot x-axis: date

Figure 3.6: Final version of the `GraphPage`. The user can click on a tab and get a preset of graphs related to the category of the tab. Here, Custom Graphs is selected.

This design gives the user a preset of graphs, making the website easy to use, and they can check the same charts as new data becomes available. It also provides the tools for a user to visualize any part of the data set they wish through the custom graphs tab.

A search box at the top of the page allows a user to select a region in the highest admin level available. When the user has selected the Search Trends tab, the app filters the available choices so only countries and regions with data are displayed in the search box. If any other tab is selected, all countries are available for selection. The user will also keep the selected tags when switching to the search trends tab, such that the user does not have to re-select regions when switching between tabs.

Once at least one country is selected and this country has subregions, a second search box appears with all the subregions within the country as suggestions. This will also happen to the subregions, and a third search box suggesting further subregions appears if admin level 2 data is available. There will be as many search boxes as administrative levels available, which is level 2 for the COD. Therefore, the user can see up to 3 search boxes where data is available.

The user can select as many regions as they want. However, the time spent generating the graphs will increase as the number of selected regions and countries increases. The user may also experience out-of-memory issues if they choose more data than their device supports. Because of time limitations, the bar race component from the Search Trend tab only supports one selected region at a time. Therefore when the Search Trends tab is selected with multiple regions, this component is hidden with a prompt notifying the user.

3.2.3 Search Trends

Google only provides search trends data for the United States, Australia, Great Britain, Ireland, Singapore, and New Zealand. Only these countries are available for selection when the search trends tab is selected. When a single country/region is selected, a bar race chart containing the 12 most searched keywords is presented to the user by default. Options such as tick duration, start date, and the number of keywords are available to adjust to preference. Figure 3.7 shows this chart with 12 of the United States' most searched words. All the available search trend keywords come from the COD. These trends are selected by Google and are all related to COVID-19.

The data for each keyword is given by the relative popularity within a geographical region. The data set from Google follows a set of rules to normalize and scale it appropriately:

1. Count the number of searches of each keyword in every region that day or week.
2. Divide this count by the total number of searches for all keywords combined in each region. This number is called normalized popularity.
3. Find the maximum value across the entire time range for each region. This value is scaled to 100, and every other value is scaled down linearly to maintain the correct ratio.
4. The scaling ratio is saved so it can be used in future releases of new data within that region. If a keyword exceeds the maximum found in step 3, the newly scaled values could be larger than 100.

The normalized popularity mentioned in step 2 is also known as the probability that a user in that region will search for that specific symptom. Since each region and time resolution are scaled with the same factor, it only makes sense to compare data within a region. Scaling factors differ from region to region, so there is no reason to compare regions together. A more detailed explanation of how the search trends symptoms data set is made can be found in Google's GitHub repository [31].

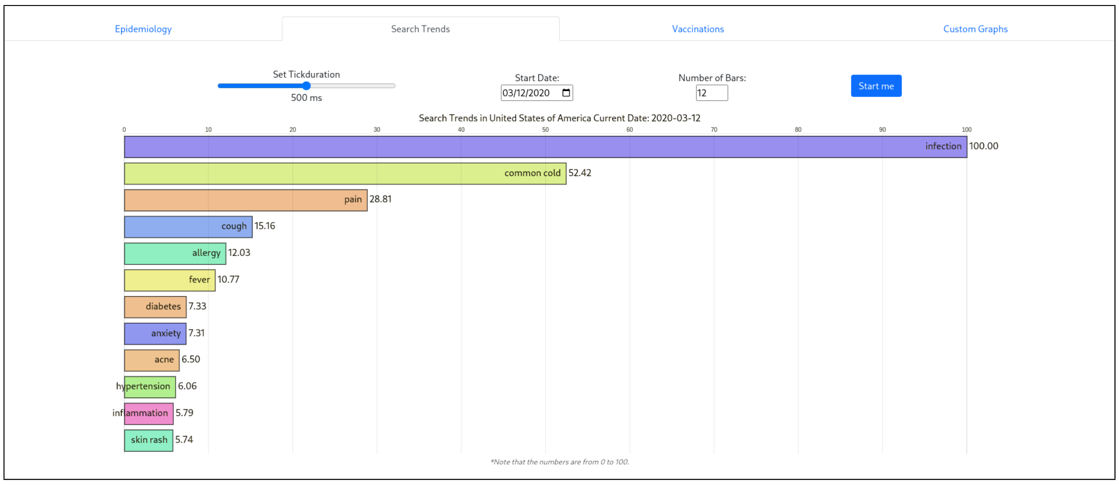


Figure 3.7: The bar race chart with the 12 most searched keywords in the US.

Further down on the page, there are multiple graphs displaying relevant search trends data. The group selected the searches which were deemed most relevant to COVID-19. These include searches such as infection, common cold, fever, anosmia (loss of smell), and more. The user will have to manually select other search trends from the custom graphs tab to view other search trends.

The search trends tab has a button to send the user to another page which displays a map of admin level 1 of the selected country, see Figure 3.8. The SearchTrendMap is not available for every country, as data for subregions in countries is only available for Australia and the US. Therefore, the button on the search trend tab for switching to this page is only visible if one of these countries is selected.

3.2.4 SearchTrendsMap

The SearchTrendsMap has a map displaying how much a search trend is searched in regions within a country. The user can use a dropdown menu to choose which keyword to display data for. Any available keyword for the country can be selected.

As seen in Figure 3.8, the map can change projection depending on the country. The projection is only changed for the United States of America. Not many countries with specialized projections are available, so this feature is not used on other pages. For

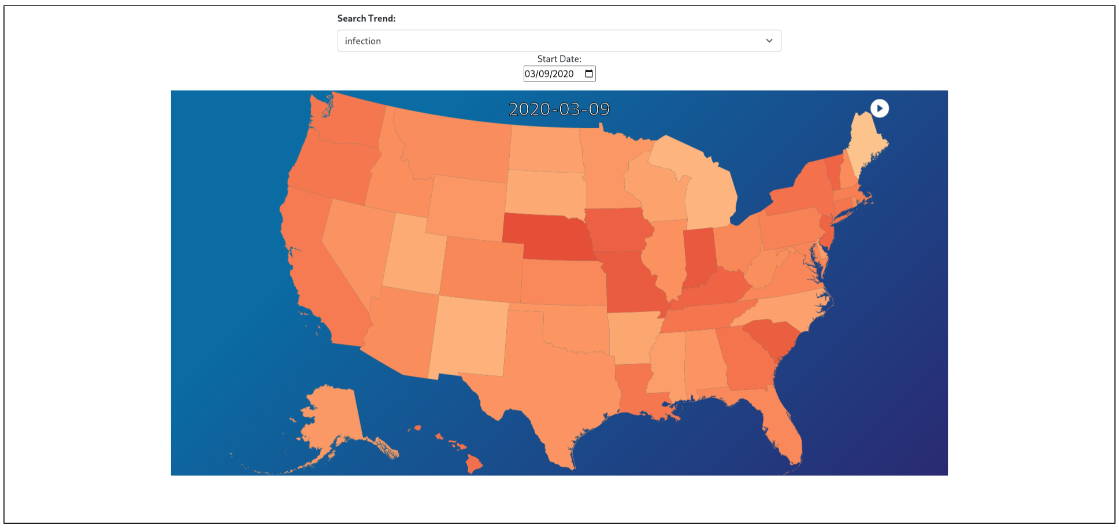


Figure 3.8: Screenshot of search trend `infection` in admin 1 level in the US.

Australia, the standard equirectangular projection is used, but the map filters out unrelated features and zooms the map, so the features take up the entire SVG area. It was also necessary to change the projection for the USA because some territories are disconnected from the mainland.

3.3 Components

This section explains the core concepts of some of the most essential components in the project.

3.3.1 MapComponent

The Map component is the most complex component of the project. Naturally, it became one of the most essential components to hide the logic internally and give a simple-to-use interface for top-level components. One could easily reuse the map in several places. It was also decided to split this component into a parent and a child component, where the parent holds and maintains the state, and the child is responsible for drawing all the geospatial features. The `MapPage` component is used in both the index page and in the `SearchTrendsMapPage` component. Listing 3.1 shows the structure of the `Map` component with some unnecessary details removed.

```
const MapComponent = ({ adminLvl , innerData = false , ...}: MapProps) => {
    // State
    ...

    // Hooks for maintaining state
    ...

    return (<>{
        data.size === 0 ? <ProgressBar animated now={100}></ProgressBar>
        :
        <  DrawMap GeoJson={curGeoJson} InnerGeoJsonProp={innerGeoJson}
        country={country} DataTypeProperty={DataTypeProperty}
        Data={data} CurDate={Date}
        adminLvl={adminLvl} height={height}
        width={width} scalePer100K={scalePer100k} />
    }</>);
}
```

Listing 3.1: Structure of the Map component. Note that the code is not a direct copy of the source code, and is only used to give insight into the component's structure

The interface is comprised of 11 parameters that can be changed to satisfy many potential use cases. Listing 3.2 shows the interface of the `Map` component with TypeScript type definitions.

```
interface MapProps = {
    adminLvl: 0 | 1 | 2,
    data: Map<string, DataType[]>,
    innerData?: boolean,
    country?: string,
    Date: string,
    DataTypeProperty: keyof DataType,
    height: number,
    width: number,
    scalePer100k?: boolean,
    loadedData: (Data: Map<string, DataType[]>) => void
    LoadData?: typeof _LoadSmallData,
}
```

Listing 3.2: Type definition of the parameters for the `Map` component.

This interface enables top-level components to not to be concerned about how the map is created and displayed. Instead, it only has to consider the size in pixels the map should be, and which data it should show. An explanation for the props can be found in Table 3.1. Note that the question mark behind a property variable means it is optional.

Table 3.1: The different props to the `Map` component with a description.

Parameter	Description
adminLvl:	The default administrative level the map should show.
data:	COVID data loaded that the map should should display.
innerData?:	Boolean that indicates whether the map should also load and display the next administrative level.
country?:	Optional parameter, if defined the map will filter out all other spatial features that does not have the selected country code.
Date:	Date that should be displayed by the map.
DataTypeProperty:	The key choosen that the map should use to select data.
height:	Height of the SVG.
width:	Width of the SVG.
scalePer100k?:	Boolean defining if the map should scale with the "population" key in the data.
loadedData:	Function for setting the data in the parent's state.
LoadData?:	Optional function that is used for loading the data to be used with the map. If no function is specified there is a default function for loading the data.

Interfaces like this one lowers the code coupling substantially. Take the `LoadData` parameter, it only requires that the function has to be of the same type as the default function used in the component. Having the function as a parameter means that the function can easily be swapped in different contexts. This pragmatic way of creating components makes it easier to test code in isolation in a unit test. It would be impossible to create anything other than integration or end-to-end tests without loosely coupled code.

The actual drawing of the map is done by the `drawMap` component. First, the `Map` component loads the necessary GeoJSON files and passes them down to the `drawMap` component in addition to the data and some of its props. Then `drawMap` component then generates the color scale to use, as previously explained in Section 3.2.1. It does this by retrieving the column `LOCATIONKEY` in the GeoJSON.

This column is not from Natural Earth but is generated by our Python script created to match the geospatial features to the correct location key in the COD. If this key is missing in the GeoJSON, the Python script was unsuccessful in finding the correct location key for this feature. The reason differs depending on the feature in question, but this is further discussed in Section 4.5.5. The `drawMap` also hosts the tooltip object, which is responsible for showing the tooltip of the map.

3.3.2 DateHistogram

The component `DateHistogram` provides a quick overview of the global situation for the selected data filter. When the filter is set to new cases, the histogram represents new cases globally for every date since the pandemic's start. When hovering over the histogram, a black line follows the mouse pointer to indicate which date that will be shown when the histogram is clicked. When the current date in the component's properties is updated, a red line is drawn to represent the selected date. Figure 3.9 shows what the histogram component looks like. Notice the black and red lines indicating where the user is hovering their mouse, and the date currently selected respectively. The component is placed at the bottom of the index page, which means it is drawn over Antarctica by default. The histogram can be seen on the front page in Figure 3.3.

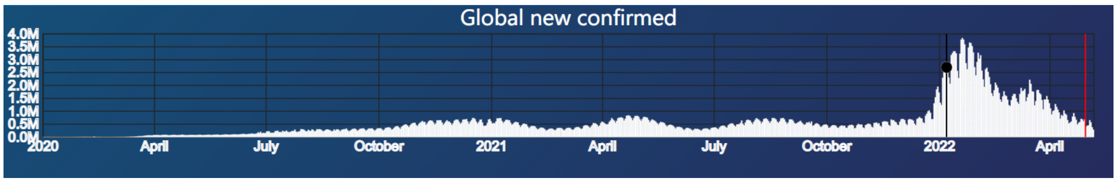


Figure 3.9: Histogram component showing the global development of confirmed cases.

3.3.3 Graphs

The different graph components are so similar that they will be discussed together in this section. Several of the graphs created took inspiration from the D3.js graph gallery, where visualizations made by others are shown with their source code. This gallery was also helpful in understanding how to use the D3.js framework.

There are some differences in the interfaces between the graphs created, but they are minuscule and are handled by the `PlotsContainer`. This component takes an array of the type `Plot` and creates the correct graphs from the array. Page components only have to keep an array of plot objects in the state to display any of the graph types. Listing 3.3 shows the interface of the `Plot` type.

```
export type Plot {
    PlotType: PlotType,
    MapData: Map<string, DataType[]>,
    Axis: (keyof DataType)[],
    Height: number,
    Width: number,
    Title: string,
}
```

Listing 3.3: Plot type which all the plots use. Code taken from `PlotType.ts`.

The `PlotType` indicates the type of graph (such as line chart, scatter), the data it should display, the properties of the axes, the size of the created SVG, and the title.

How the different graphs create their illustrations differs somewhat. For example, the `linechart` component needs a line generator to be able to create the lines used in the visualizations, while the scatter plots use the standard circle SVG element. The tooltips are also different depending on the type of plot created. Scatter plots use react-bootstrap's `OverlayTrigger` component, which creates a tooltip when the mouse hovers over a point. This component would not work for the line chart without major adaption. This is because with the line chart it should show the data for all of the lines at that date. For this reason, the line chart uses a function called `updateTooltip()` to handle hovering on the line chart. The two different tooltips and graphs are shown in Figure 3.10.

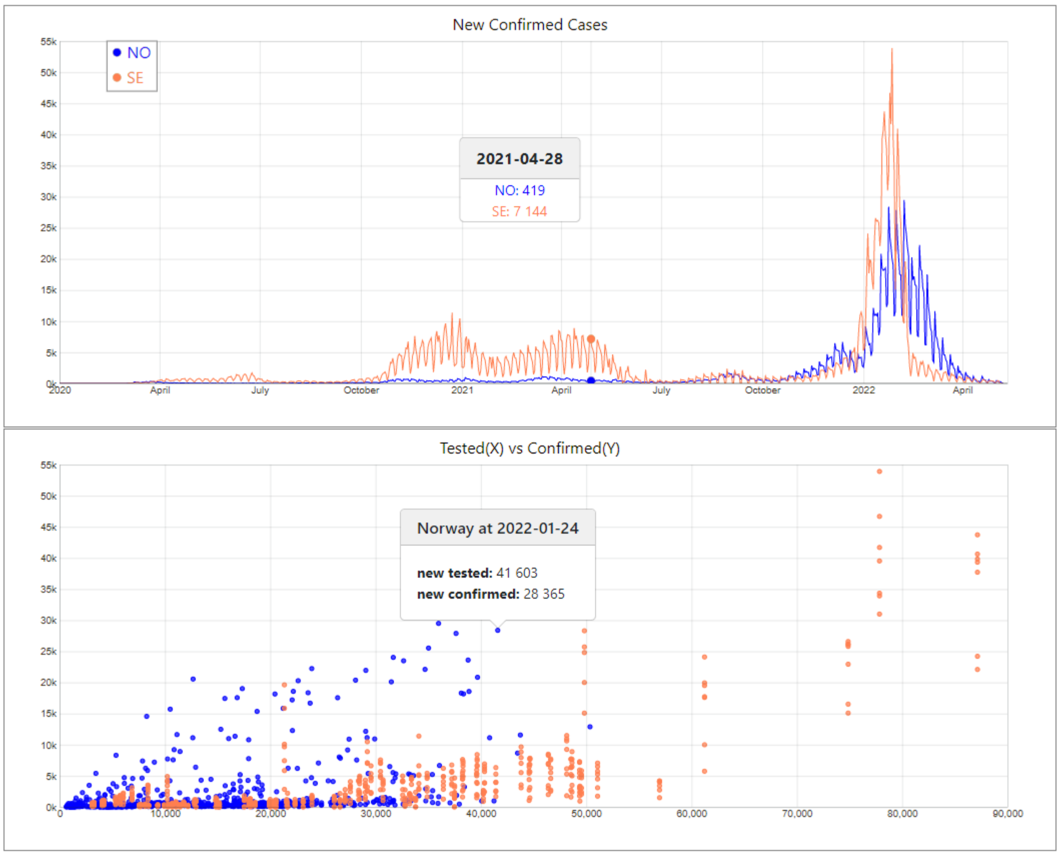


Figure 3.10: Examples of the different tooltip styles for the line and scatter plots.

The `linechart` component uses the event's position and then calculates the date from the inverse of the xScale function (the function takes a date and returns a position in pixels) to find the date closest to the mouse position. It then draws points for the corresponding date for each of the selected regions, and it creates a div displaying the regions, values, and date.

3.3.4 SelectCountry

The `SelectCountry` component uses the React package `react-tags-autocomplete` [32]. The package comes with standard functions to use when adding and deleting autocompleted tags from the search bar. The component utilizes a type called `TagExtended`, which is an extension of the default `Tag` type provided by the `react-tags-autocomplete` package. All tags must have a unique id and name, where the name is what is shown to the user.

The component uses the index file from COD to retrieve all available regions. This data is then structured in a hashmap, where the key is the `location_key` from COD, and the value is a custom object. This object contains a unique id, the location key, an array of location keys for all the subregions, and the location's name. The reason for having an array of subregions is because the `SelectCountry` component only allows the user to select subregions from regions selected. This approach was chosen to not overwhelm the user with thousands of admin level 1 and 2 suggestions. It also allows the user to know that the regions selected are tied to the already selected regions. The id and the name are used for the`react-tags-autocomplete`'s tag type.

Figure 3.11 shows an example where the US has been selected together with California and Alameda County. Note that in the suggestion list, Alameda has been removed since it is already selected.

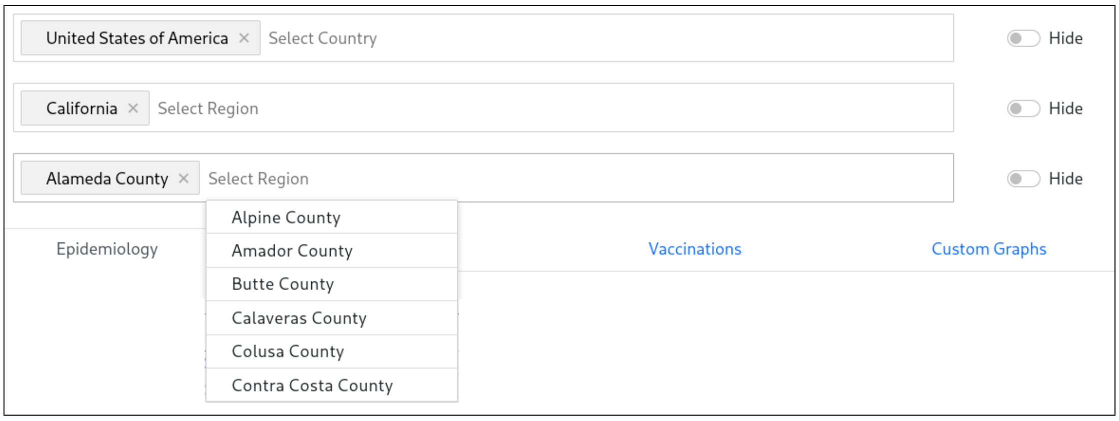


Figure 3.11: Select Country component with selected locations for all available admin levels.

Since the `SelectCountry` component handles up to three different admin levels, the component needs to keep track of all the admin levels and the corresponding data. The component does this by having a hashmap in the state where the key is the admin level, and the value is an object used to keep track of the current selection for that admin level. This object has all available suggestions (data from the index file), active suggestions (all the suggestions without the chosen tags), tags (the current selections), and a boolean determines if the tags should be passed to a parent component as they are selected. Listing 3.4 shows the state and the interfaces/types of the component.

```
type AdminlvlEntry = {
    tags: Map<string, Tag>,
    activeSuggestions: Map<string, IMap>,
    hideData: boolean,
    allSuggestions: Map<string, IMap>
}
interface IMap {
    id: number,
    locationKey: string,
    children: string[],
    name: string
}

const [adminLvls, setAdminLvls] = useState<Map<number, AdminlvlEntry>>(
    new Map());
```

Listing 3.4: The state and used interfaces and types for the `SelectCountry` component.

The only state is a hashmap with the admin level, and its corresponding entry. Listing 3.5 shows two examples of what an `IMap` object may look like.

```
{1, "NO", ["NO_01", "NO_02", "NO_03", ...], "Norway"}
{2, "NO_11", [], "Rogaland"}
```

Listing 3.5: Example objects of the IMap interface.

The tabs displayed on the graph page need a unique key property. This key is passed to the `SelectCountry` component and is used to know if the `SelectCountry` component should filter to countries that have search trends data. Since there are only six countries, and the group believe these are not subject to change, they are hardcoded in an array. Only these countries are suggested if the key is the one from the search trends tab.

3.4 Data Handling Approach

The following section explains the solution for how data is being handled in the application.

3.4.1 Google Cloud Storage and BigQuery

The final approach to how data sets are reduced in size for the front page is done using Cloud Storage and BigQuery. The latest epidemiology CSV file is uploaded to a custom-created bucket called `covid-minimized` in Google Cloud Storage. The data set is transferred from the bucket into a table in BigQuery, where it is possible to query the table using SQL, similar to how one would query a database. Listing 3.6 is one of the queries used for filtering out everything except admin level 0 data and their confirmed cases from the epidemiology table. The demographics table was also included to allow the map to display the data per 100k population.

```sql
SELECT
    date,
    epidemiology.location_key,
    new_confirmed,
    population
FROM
  `bachelor-thesis-345612.covid_epi.epi` AS epidemiology
INNER JOIN
  `bachelor-thesis-345612.covid_epi.demographics` AS demographics
ON
    demographics.location_key = epidemiology.location_key
WHERE
    length(epidemiology.location_key)=2
```

Listing 3.6: SQL query for retrieving all countries' newly confirmed cases.

Every location key for admin level 0 will always be two characters long, which allows the condition `length(location_key)=2` to be used to filter out other admin levels. The result of the query is saved to the bucket in the Cloud Storage and, from there, downloaded as a CSV file. On the 23rd of April, the final file was reduced from 11 413 228 to 190 813 rows, which is equivalent to a reduction of 98.32% rows. The number of columns in the CSV files is constant, and the percentage of reduction is expected to be similar at later dates. These smaller CSV files are accessed with an API call, as seen in Listing 3.7.

```typescript
const _LoadSmallData = (datatype: keyof DataType="new_confirmed",
   locations: string[]=[]) => {
    if (datatype === "new_confirmed" || datatype === "new_deceased") {
        return new Promise<Map<string, DataType[]>>((resolve) => {
          csv("https://storage.googleapis.com/covid-data-minimized/"+
   datatype+".csv").then(d => {

            // handle data
                ...

                resolve(data)
            })
        })
    } else {
        return LoadDataAsMap(locations, new Map())
    };
}
```

Listing 3.7: Function which makes the API call to reduced data sets.

Currently, reducing the data sets with BigQuery is done manually only for new cases and new deceased, as those are the most relevant data types to display on the map. The sidebar does allow filtering of other data, such as new hospitalizations, tested, or vaccinations. When one of the other categories is selected, the LoadDataAsMap component will use the Aggregated data table from Google instead. The downside with this is that loading is much slower due to the much larger unreduced data.

Some authentication is required to allow interaction between a client and a Cloud Storage Bucket. Today's modern browsers use a same-origin policy, which means a website is not allowed to access resources outside its original domain without the external domain explicitly allowing the original domain in its `Access-Control-Allow-Origin` header. This header tells the browser which domains are allowed to access resources. There are three ways this can be defined: "<clients>" which is the allowed clients, "null" (not recommended), and "*" (wildcard) allowing any origin to access this resource. In the header, one will also find the `Access-Control-Allow-Methods` for what methods the origin is allowed as well as the `Access-Control-Max-Age` (in seconds) for how long the browser can keep a preflight of the external domain.

A preflight is a request just checking if the origin is allowed to access the resource without actually requesting the content. If the preflight succeeds, the browser will request the data. Each bucket in Google Cloud Storage has its own settings for Cross Origin Resource Sharing (CORS) where the header can be specified. These settings were necessary to create for the project to be able to use Google Cloud Storage. If it were not explicitly defined, there would be a CORS error. A JSON file was created to correct this and has been copied into the GitHub repository. Listing 3.8 shows the settings specified.

```
[
  {
    "origin": [
      "https://bachelor-group.github.io/",
      "http://localhost:3000"
    ],
    "method": [
      "GET"
    ],
    "responseHeader": [
      "Content-Type"
    ],
    "maxAgeSeconds": 3600
  }
]
```

Listing 3.8: The CORS settings for the bucket in Google Cloud Storage.

A command-line tool called `gsutil` [33], which is a python script, was used to update the JSON file at Google. This tool updated the settings file at Google, and the browser was now allowed access to resources from the Google API.

3.4.2 Histogram Data

Most of the data for the web application is reduced using BigQuery. An exception to this is the data for the histogram. For the categories new confirmed and new deceased, a python script downloads the reduced files from Google Cloud Storage. The script goes through them and calculates the total cases for all countries over each date. Results are saved to a file and then automatically uploaded to Cloud Storage.

The hospitalizations, tested, and vaccinations categories do not have a sliced version in the `covid-minimized` bucket. As described in Section 3.4.1, when these categories are selected, the `Map` loads the data from the aggregated data tables from the COD. The histogram uses that same data set and does the same calculation as the python script. However, it is done on the client-side.

3.5 Hosting

The group decided to use GitHub due to prior experience of using it, and its popularity among developers. Some time was spent reviewing different solutions to hosting. Heroku, Amazon Web Services, Microsoft Azure, Firebase, and the UiS Pitter lab were all considered. However, since the project's repository is hosted on GitHub, using GitHub Pages was a simple step of using a preexisting workflow and, therefore, was the chosen platform. GitHub Pages is also a free hosting platform with limits well within the scope of the project.

With GitHub Actions, it is possible to define workflows that ensure the website is updated on every push on the main branch or pull request with the main branch as the base. The deployed version of the source code is in the branch `gh-pages` and can be found at `https://github.com/bachelor-group/bachelor-group.github.io/tree/gh-pages`.

Every time a new commit is pushed to the main branch, all tests are run. These tests are mainly testing that the components mount correctly. If all tests pass on a continuous integration server hosted by GitHub with a fresh install of node.js, the changes are deployed to the web application.

Chapter 4

Discussion

This chapter discusses earlier approaches used before the final solution was implemented. It also discusses why some of the technical decisions were made in more detail.

4.1 Adapting the Thesis Proposal

As time passed, it was clear that the project's domain had begun to expand further than the original thesis proposal had initially suggested. The original thesis proposal focused primarily on search trends and their correlation to newly confirmed and deceased cases of COVID. The web application still provides a good overview of search trends and focuses much more on the general evolution of epidemiology and vaccinations. Graphs are not only showing the correlation of search trends to COVID cases but include more categories, such as vaccinations and epidemiology. The thesis proposal presented in chapter 1 is a slightly altered version of the original proposal.

4.2 Technology Analysis

From one of the first meetings with the thesis supervisor, it was clear that D3.js would be a good option to use for visualizing the data. The supervisor had some prior experience after using it for his thesis. There was not much discussion for this decision, as there was no prior knowledge of any other similar libraries. In addition, the graphs we envisioned to have in the final application were close to the ones found on the D3.js graph gallery.

The main reasons for choosing React was its popularity among developers, and that we wanted to learn it. We had previous experience using the framework VueJS, which is similarly to React. Therefore, React was deemed a suitable framework to use in this project.

Considering React is also the most used JavaScript framework [10], it means there is likely more documentation of it together with D3.js than there is with, for example, VueJS.

There are several reasons why using TypeScript is a good idea. For us, it was mainly the fact that it is a statically typed language. Knowing what types are expected makes it easier for people to work on different parts of the application, and to understand and develop another person's code. Bugs are also much easier to spot, which only adds to the advantages of TypeScript.

The main reason GitHub Pages became the final solution to hosting was that it fit our criteria for a hosting solution. GitHub pages is a completely free solution with some usage limits, such as 1 GB max repository size and a soft 100 GB bandwidth limit per month [34]. Since the project is on a relatively small scale, it does not exceed these limits. Other platforms such as Amazon Web Services and Microsoft Azure would likely also have worked. However, the free student credits provided by these services might not have been enough to match the project's lifetime. GitHub Pages is also straightforward to set up, and there are only a few specific naming conventions to follow when using the GitHub Pages organization site [35].

4.3 The war for the DOM

One of the biggest challenges with this project was deciding the amount of control to give to each framework. Both D3.js and React want to have absolute control over the DOM, and making the frameworks work together efficiently is no easy task. As explained earlier (Section 2.2.2), React will only update components on state changes to maximize performance. This lifecycle did make animating with React a more strenuous endeavor.

For example, one can look at the `BarRace` component on the search trends page. This component could technically be animated with React by changing the parameters of the array of rectangles, for instance, the width and color. React can also change the texts and numbers, so React can do most of the things needed for the desired result. However, React can not uniquely identify each rectangle and does not intuitively know the previous state. Therefore, React does not know about a rectangle's previous position, making the animation of the vertical position change for `BarRace` near impossible. One could look at React as a discrete system where react only knows about the current state, and there is no overlap from the previous state. React would instead think the same rectangles are in the same location with different widths and colors. This way of updating merely made React change the color of each rectangle, and the vertical flipping animation in the result was also cumbersome to create.

Because of this, it was decided to give D3.js almost all control of the DOM. The animation of the `BarRace` component and allowing users to interact with maps and graphs is all handled by D3.js. This information is not kept in the state of a component to stop react redrawing and ruining the animations. React instead handles the structure of the webpage, i.e., where the components are placed. This approach helped a lot in making animations with D3.js in no time. There were still problems to overcome with this method. One problem that took a considerable time was some of the callback functions used. The main problem is taking a value from React's state and then using and updating it in the callback function. As an example, one can look at the `Animator` component for the map. Listing 4.1 shows the final implementation of the animate function.

```
async function Animate() {
    // animation is already playing
    if (ticker !== undefined) {
        ticker.stop();
        setTicker(undefined);
    }
    else {
        let cursor = FindDateIndex(startDate) - 1
        let tickerTemp = interval(e => {
            if (cursor >= barsData.length - 1) { tickerTemp.stop();
   setTicker(undefined); return };
            cursor = cursor + 1;
            updatePlot(cursor);
        }, tickDuration);
        setTicker(tickerTemp);
    }
}
```

Listing 4.1: Example from `BarRace` component showing interval callback.

When the animation starts, the current date resides in the parent component's state, but as the animation continues playing, the current date needs to be updated every time the callback is called. The current date will be a variable in this callback, and if implemented in the wrong way, the date would be the same every time the callback is called.

4.4 Design Changes

The original idea was to use the data set's categories, such as epidemiology and vaccinations, and have a link in the navbar directing the user to pages showing data related to each category. Initially, the navbar would have had links for a page with charts for the different data categories, and links to maps with the corresponding data. The original concept can be seen in Figure 4.1. We later decided against this approach to improve the user experience, and increase performance, see Section 3.2.2.

It was decided not to have different types of graphs on individual pages, but to rather have one page with tabs for some data type presets instead. Since this was decided upon later, early versions of the graph page were split into several pages.

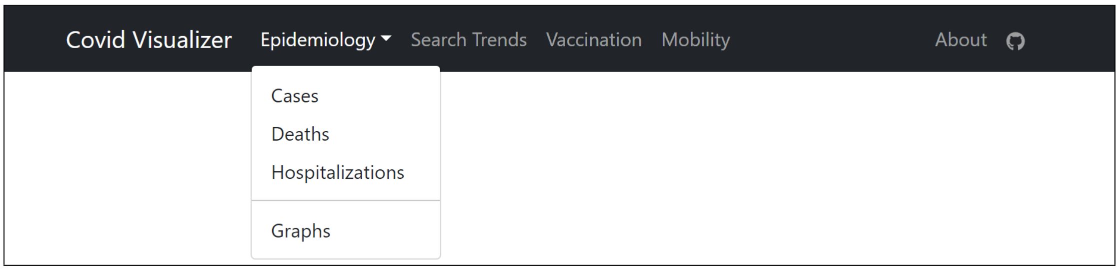


Figure 4.1: Early version of the navbar showing the original concept of sub-menus.

The reasoning behind the changed approach was the thought that users who wanted to see data in graphs were more likely to go to several of the pages with the different data categories. To have the user navigate between pages would not have been an optimal approach, as pages would have had to either reload the data, or keep it in local storage. In addition, some of the state would have had to be sent in the URL to indicate what a user had selected previously.

If the data had been reloaded on page changes, it would put significant strain on the server and client, who would exchange the same data several times, which would have been inefficient. It is crucial to be efficient to decrease the environmental impact of the application and improve user experience. For example, if a user wanted to compare Norway and Sweden, they would have to choose these countries, but if a user changed the page, this would have had to be encoded in the URL if these selections were to be retained on navigation.

Instead, it was determined to keep users on the same page and allow them to choose between some preset graphs for each category, and an option where the user is able to choose any parameters from the data set.

4.5 Data Handling

Deciding how to handle the data from enormous JSON and CSV files is crucial to optimize performance. Some of the files from Google contain up to 11 million objects, and are growing every day. As mentioned in Section 2.3, the data sets from google can only be filtered by location, category or to the latest data. These filters were not sufficient for this application's use case. Therefore, we decided to copy data from Google and tailor the filters and slicing to this project.

4.5.1 Slicing Data

As described in Section 3.4, our solution for handling data was to slice it into smaller pieces and then serve it. Python scripts were used to reduce CSV files before the final decision landed on using BigQuery and Cloud Storage.

As previously stated in 3.2.1, the map displays data one week back in time. Having a buffer of seven days is mainly done for two reasons. The first is that the CSV files which provide the data are uploaded manually, so lagging one week behind provides a buffer such that the data does not have to be updated daily. Another reason is that not every country has data at the latest possible date in the data set provided by Google. One week back in time virtually guarantees that all countries have data.

Early Python Script for Slicing

An earlier approach was to use the index data set from Google and retrieve every location key, then use them to fetch the aggregated file filtered by those location keys. After fetching the aggregated file for each location, they would be split into even smaller files, such as `cases.csv`, `deceased.csv`, `vaccinations.csv`, `epidemiology.csv`. The files were saved in a structured way, making it easy to retrieve them. See Figure 4.2 for an example structure.

These files contain the bare minimum of what is required to show data. Both `cases.csv` and `deceased.csv` would be used for the front page map, while the others would be used for the Graph page; one file for each tab.

The file structure would have been uploaded to Cloud Storage, so any needed location could easily be fetched with an API call. One of the issues with this approach is uploading to Cloud Storage. There was an extreme amount of files that had to be uploaded, which Cloud Storage did not seem to handle well. Uploading all the files took well over an hour. It did not help that downloading the files also took a long time.

```
│ AD
│  ├── cases.csv
│  ├── deceased.csv
│  ├── vaccinations.csv
│  ├── epidemiology.csv
│  └── ...
├── NO
│  ├── cases.csv
│  ├── ...
│  ├── 01
│  │  ├── cases.csv
│  │  └── ...
│  ├── 02
│  │  ├── cases.csv
│  │  └── ...
│  ├── 03
│  │  ├── cases.csv
│  │  └── ...
│  └── ...
├── US
│  ├── AK
│  │  ├── 02013
│  │  │  ├── cases.csv
│  │  │  └── ...
│  │  └── ...
│  └── ...
└── ...
```

Figure 4.2: File structure for aggregated files filtered by location.

This approach would have been especially beneficial for the graph page since only one location is added at a time. However, this was not the final approach, as the whole process was too time-consuming. Cloud Storage was used in a different way together with BigQuery, as previously explained in Section 3.4.1.

Concurrency

While the Python script is running, most of the time is spent waiting on network calls to each location. Doing this for around 23.000 locations takes a very long time. Concurrency comes in handy to reduce idling times. We can do multiple network calls simultaneously using multiple threads, saving substantial time waiting for HTTP requests.

The Python script was benchmarked with and without concurrency to see the difference in performance. The results were very much as expected; Using multiple threads was much faster. Listings 4.2 and 4.3 show the summaries of the benchmarking outputs. Full benchmark outputs can be studied in Appendix B. Bear in mind that only the first 200 lines of the index file were used during the benchmarking.

```
 1 200 lines, 198 regions, 2 empty lines:
 2 16 threads:
 3 Command being timed: "python ./Python/generate_data.py"
 4   User time (seconds): 24.39
 5   System time (seconds): 7.05
 6   Percent of CPU this job got: 84%
 7   Elapsed (wall clock) time (h:mm:ss or m:ss):  0:37.40
 8   ...
 9   Maximum resident set size (kbytes): 2229780
10   Average resident set size (kbytes): 0
11   Major (requiring I/O) page faults: 0
12   Minor (reclaiming a frame) page faults: 803764
13   Voluntary context switches: 310249
14   Involuntary context switches: 6042
15   ...
```

Listing 4.2: 16 threads, concurrent python script.

```
 1 200 lines, 198 regions, 2 empty lines:
 2 1 thread:
 3 Command being timed: "python ./Python/generate_data.py"
 4   User time (seconds): 31.15
 5   System time (seconds): 6.53
 6   Percent of CPU this job got: 36%
 7   Elapsed (wall clock) time (h:mm:ss or m:ss):  1:42.38
 8   ...
 9   Maximum resident set size (kbytes): 2229768
10   Average resident set size (kbytes): 0
11   Major (requiring I/O) page faults: 0
12   Minor (reclaiming a frame) page faults: 825778
13   Voluntary context switches: 26251
14   Involuntary context switches: 2623
15     ...
```

Listing 4.3: 1 thread, non concurrent python script

Elapsed time for the concurrent version is 37 seconds, while the non concurrent script takes 1m 42s (see line 7 in Listing 4.3). The concurrent version was almost three times faster than the serialized version.

BigQuery and Google Cloud Storage Reasoning

It was decided to only use the CSV format, mainly because it is available for all categories, while only a limited number of categories provide a JSON version. CSV files are also considered to be faster than JSON. Less bandwidth is needed, and data processing is quicker due to not having to interpret JSON syntax [36].

We contemplated implementing a back-end with a database that would be optimized for this use case. However, while researching, we were unsuccessful in finding a solution that was able to support the amount of data needed within the project's price range. While looking for a suitable place to store data, we discovered BigQuery and Google Cloud Storage.

BigQuery is in no way trying to replace a relational database. However, it is convenient to use when there is no need for the basic Create, Read, Update, Delete (CRUD) operations [37]. Because Google generates the data, the project did not need CRUD functionality, and because of the convenience of BigQuery, it was chosen over a traditional database. The app will never alter or delete information from the data sets, only retrieve them.

The Google Health team gave insight into their approach through correspondence over email. Their approach was to slice their data and serve it with the help of BigQuery. Google mainly uses Google Cloud Storage, which can automatically handle decompressive transcoding. Decompressive transcoding allows for storing compressed versions of files in the Cloud Storage while at the same time serving the file to users without compression [38], thus reducing costs. For Decompressive transcoding, files must be compressed with gzip. However, it is faster while compressing and decompressing [39]. Google also uses BigQuery to support SQL and enterprise access to their data sets.

With all this in mind, it was concluded that there was no need for a database, and a similar approach to Google was used.

4.5.2 When to Load Data

One crucial decision when optimizing performance is choosing when to load data. Loading all the data when opening the page, or loading piece by piece when it is needed to display whatever the user is requesting are the two approaches that can be taken.

Immediately loading all the data allows us to keep it in memory, which is beneficial for applying different filters and changing visualization methods. However, this method would prolong the initial load, and all of the data may not be needed; resulting in wasted load time. When to load is a trivial question for many applications, where differences are minimal. The answer is not as simple for a project where the amounts of data could take many seconds to load.

When loading at request, it is ensured that all the loaded data is used. This approach makes the initial loading quick, but the site's overall experience would be lowered as a user must wait on data being loaded on most event triggers. It is crucial for the user experience to strike the right balance between initial load time, and load times at event triggers.

Take, for example, the map of the index page. The GeoJSON at admin 0 has to be loaded initially, as the `Map` component has nothing to display without it. Most people would probably agree with this approach, but the same may not be said for loading the admin 1 data.

If users click on nations in most cases, it is better to load this data initially and reduce the time spent on the click event. However, if it is a rare occurrence for users to use this feature, it would be better to wait.

A small test was conducted with the help of fellow students. They were asked to use the website as we observed their actions on the site. The result of this test was that most of them clicked on some countries. With this, it was concluded that loading GeoJSON for both admin level 0 and 1 on initial load was the best option.

It can also be debated for each country's data. The page needs to initially load data for every nation at the date shown initially. It does not need other dates, but a user would have to wait each time they change the date. It is reasonable to believe that changing the date is a common incident, and therefore data should already be loaded beforehand so a users do not have to wait for more data to be loaded.

Therefore, the approach chosen was to load data for all the dates when the map page is accessed.

This approach ensures that animating between dates goes smoothly, as the client does not have to wait for each date to be loaded sequentially.

4.5.3 Choosing Data Structure

When working with CSV files, the files are often converted to an array of objects. Each element is an object containing keys that correspond with all the CSV file columns. The most naive implementation is keeping the data as an array throughout the project. This naive implementation was the approach used in the early stages of the project. We knew early on that this was not going to be the final data structure, but since many details of what visualizations were going to be in the application were yet to be decided on, it was kept in this naive state initially. This naive implementation was used as a proof of concept, getting the application to a working state, where efforts could be spent on creating the visualizations needed for the final project. As the project matured and more of the specifics were decided upon, it became clear that choosing a proper data structure had to be done.

One example, which will be explored further in this section, is the `Map` component and its data structure. At the beginning of the project, the `Map` used aggregated tables to load data. Since the `Map` needed to load data for every nation, it would have to do some 200 API calls for data. Furthermore, since this was an aggregated table, a lot of the data columns transmitted would not be used. The aggregated tables were extremely slow and wasted a considerable amount of memory and network resources. This implementation was kept for around for half the project's development time, as it was difficult to chooe how to store and the load data; as explained in Section 4.5.

Once the final decision landed on a custom sliced CSV file with the data for all admin 0 locations, a proper data structure could be determined. The problem with using an ordinary array is that the search time complexity is `O(N)` where N, in this case, is the length of the data. With 195 countries in the world, and each country has data points for each day back to the first of January 2020, it would be (as of writing) around 170 000 objects to search. In addition to that, a map has to find data for every geospatial feature shown. If one is not careful, the time complexity quickly grows to be non-linear. With the use of arrays, one could at best have a search complexity of `O(N)`that is, if you create a key-value pair containing all the countries and corresponding data points as the program searches through the array. This search would have to be run every time the map was updated, for example, when changing the date.

This data structure can be improved considerably. The final data structure used was JavaScript's `Map` type. This type is JavaScript's way of using hashmaps. Objects in JavaScript allow for key-value pairs, so there are only minor differences between the hashmap and an ordinary `Object`. The hashmap structure used had the key set to the `location_key` from the COD, and the value as an array of the data associated with the key. The program still has to search through the list of data, so it still has a time complexity of `O(N)`. The significant difference is that `N` now refers to the number of days since the first of January 2020. That is not to say that this is the most efficient data structure, but it is a significant improvement over the naive approach.

4.5.4 Choosing the Data set

It was no surprise that the application needed a significant amount of data to be of value. Luckily, since the start of the pandemic, most countries have been more than willing to share their data so the world collectively could learn more about the outbreak. The biggest problem is that there is no consistent structure between published data from all countries, which made compatibility between nations a problem. Google provided one of the most comprehensive and well maintained data sets with a consistent structure. Meaning one can quickly get the data one wants from any file at any admin level consistently. We were initially introduced to the data repository through the project's supervisor. This application was always intended to show search trends during the pandemic. Since Google is the search engine today with the biggest market share [40], it made sense to use their data repository to get access to the most accurate data. These benefits were the reasons why COD was chosen as the application's data set.

4.5.5 Compatibility Issues Using COVID-19 Open Data Repository and Natural Earth

During the project's development, compatibility issues arose between the COVID-19 data set and the geospatial data from Natural Earth. It primarily came about as administrative level 1 was introduced into the application.

The first thing to note is how the files from Natural Earth differ depending on the administrative level used. The main issue from Natural Earth was the inconsistent naming and generalization of properties. Take something as simple as retrieving the name of a feature. In the file for admin 0, one can get the name in the following way `feature.properties.NAME`. If instead, one wanted to get the name from the admin 1 file, this would be the way to get it `feature.properties.name`. This means that in the project, one needs to keep track of which file is currently worked on to retrieve the information correctly. It was decided that it was necessary to introduce a completely separate object to handle the differences between these files. This object would also handle formatting the returned strings into the strings used for requesting data from COD but was replaced by adding a location key column in the GeoJSON using a Python script.

The name field is a simple example of inconsistent naming, but a more complicated case exists with location codes. Location keys are more complicated because of the different formats of location codes used in COD. Since the Google COD uses different standardized codes depending on the country, it is not as simple as getting a single property from the GeoJSON. It may be that the data can be found using ISO codes, but it is also possible that the code is in the NUTS, FIPS, or postcode format on Google COD. These codes may or may not be available in the Natural Earth data set. Therefore, it is not guaranteed that a location can be connected to the COVID data. This is one of the situations where the maps may display its magenta color, since it was unsuccessful in getting data for this feature.

To give a concrete example of the use of different codes, one can look at Sweden's admin level 1 features. Sweden is a part of the EU, and thus, also the NUTS standard. Therefore, Google uses NUTS codes as the `location_key` for Sweden. This is one of the codes that Natural Earth does not provide in their data. All the available codes for Stockholm county are listed in Listing 4.4. Because the NUTS codes are missing, connecting any feature from admin level 1 to the Google COD data is impossible without using an external resource or searching for the name in Google COD's index file and hoping they match. The `location_key` Google has for Stockholm is the key `SE110`.

```
properties: {
    adm1_code: "SWE-194",
    code_hasc: "SE.ST",
    fips: "SW26",
    gn_a1_code: "SE.26",
    gns_adm1: "SW26",
    iso_3166_2: "SE-AB",
}
```

Listing 4.4: All relevant code fields found in the properties for Stockholm county

The use of different codes was one of the reasons efforts to develop a separate python script were started to solve this problem. This script was to go through the GeoJSON and try different combinations in an attempt to find a `location_key` match in Google COD's index file, or worst case, use the feature name to try and find a match. After a suitable match was found, the script would add a column in the TopoJSON with the correct key. This was to massively improve the user experience of maps at admin level 1. Unfortunately, after some initially good results, it was later deemed that it was not possible to fix many of the compatibility issues. Therefore, the script was changed to see if the location key gathered from Natural Earth existed in the COD and if so, it would give this key in the added column. A Major reason for scrapping the original python script was the inconsistency of what subdivision was used in the Natural Earth data. Some countries were in administrative level 2 even when the level 1 map was used. An example of this can be seen in Figure 4.3, where the difference between admin level 1 and the project's map is shown. The application can therefore not know if a feature is administrative level 1 or 2 based on the map data used, so the app must combine the different location keys in the GeoJSON into the correct `location_key` for COD.

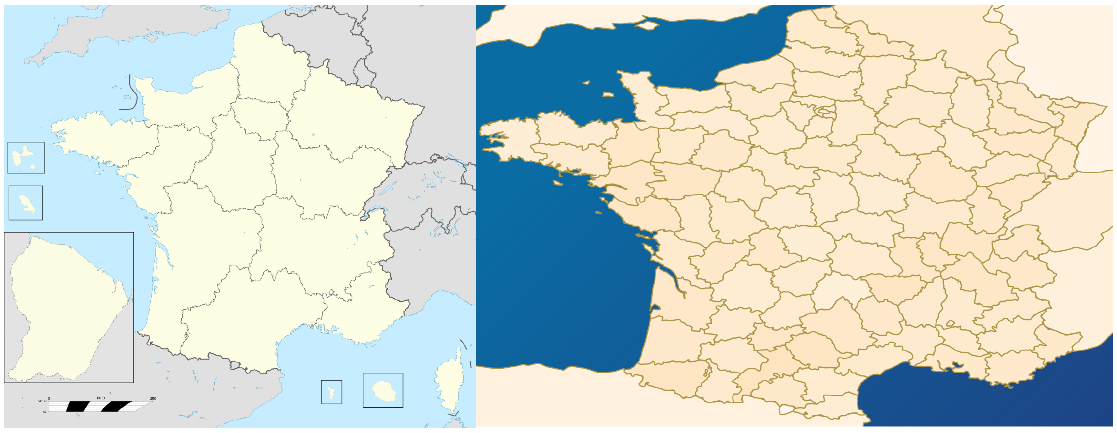


Figure 4.3: Left picture [41] shows France at administrative level 1. The picture on the right is a screenshot of the project's map and shows France in administrative level 2 called departements.

The final reason was the inconsistency in the divisions of countries between the two data sets. One of these noted early in the project was Natural Earth's division of Norway. In 2018 the provinces of Norway changed, as some of them were combined, this is reflected in the COD but has not been updated in Natural Earth. This meant that there was no data for most of Norway's features, as the old divisions are not present in COD. Thus, these features is shown as magenta on the map, as the app is unable to retrieve data for the old provinces.

4.6 Discussing Maps

This section discusses the decisions around the map, such as map type, projection, and color scale.

4.6.1 Choropleth Map

One drawback of using a choropleth map is that it may be hard to differentiate between the colors of two regions. For this reason, the tooltip for the map was created to show the raw underlying data for each region. In addition, choropleth maps emphasize larger geographical areas, and the zoom functionality is there so the user can focus on smaller regions if they desire. These features do not resolve the drawbacks of choropleth maps, but should help mitigate them.

4.6.2 Projection

When using maps, one must always decide which projection best suits the project. It is impossible to display a sphere onto a plane without distortion [42]. When showing a map on a 2D surface, it must be decided which distortion is the most acceptable for the use case.

This projection may seem slightly off for many people that are used to seeing the Mercator projection [43], which is the most commonly used projection. The Mercator projection was deemed unsuitable for this project because it drastically distorts the area. This distortion makes countries closer to the equator seem smaller compared to countries closer to the poles. The equirectangular projection's distortions make it unsuited for navigation, but it is a simple projection that is commonly used in thematic mapping [44]. These were the reasons behind the choice of a equirectangular projection.

4.6.3 Showing the True Colors of the World

Coloring the world seems like a straightforward problem, but as with projections, there is no way to color the world to satisfy all use cases.

The most important question that needed to be answered was what the purpose of the map was. Three of the purposes we contemplated were to let the consumer find patterns in the data, show how the pandemic developed from start to end, and show where the pandemic was worse day by day. The purpose would establish how the color scale should be defined. The color scale is the function that takes a domain that is the numbers from the data and returns a value that is the corresponding color. New confirmed cases are used as an example in this section, but the idea applies to any part of the data.

The simplest way to define the color scale was to find the max value of new confirmed on a single date throughout the entire pandemic and use this as the app's max value. This approach would misrepresent smaller nations that never could get close to the same number of cases as bigger nations like the United States of America, France, Germany, etc. Even if the effects of COVID on the smaller nations had had a more significant impact on them, the number of cases would remain small in comparison. This flaw is why the app early on used population to calculate the number of confirmed cases per capita.

The problem then became outlier values in the smaller nations; creating massive cases per capita. This, in turn, made the larger nation seem less important, as they would rarely close to the outliers. We had to question what is more representative of a nation's

COVID situation; the number of cases, or the number of cases per capita. It was believed that the per capita value was the better alternative, and it was chosen to use this for the map coloring. When calculating the max value, it was decided to reduce the impact of smaller nations' outlier values by not including nations with a population of less than one million.

We also experimented with using the mean value per capita and a calculated standard deviation for each date. It gave some pleasing results, and the hope was that using the per capita value would be close enough to a bell curve that the scale could be the mean plus/minus two standard deviations. After some experimenting, it was found that three standard deviations plus the mean for the maximum value and 0 for the minimum gave the most interesting result. This, however, removed the connection between the dates; colors from one date to the next could not be compared anymore, as the scale changes every day. It did show the worst areas for each day well, and made it easier to spot patterns, but it was decided not to use this color scale for its lack of continuity.

Chapter 5

Conclusion and Future Directions

In this chapter, suggestions for future improvements are proposed. Finally, the project is summarized, and conclusions of the final product and project are discussed.

5.1 Future Directions

The improvements presented here are either new features that there was no time to implement, or improvements to features already implemented.

- **Rolling Average:** Almost all the visualizations display the raw data. This is not a problem as it is interesting to see the erratic behavior of the COVID-19 outbreak, and how things change daily. There should still be an option to see data over a longer period. This feature can be created by using a rolling average for the last seven days. The Google Cloud Storage solution should be able to support the extra data. Therefore, this could be implemented by letting a python script calculate the values and store them in the cloud.
- **Clustering:** The project's supervisor has previously worked on clustering countries together based on different many COVID data types. He put forth the idea of visualizing this in the application. However, there was unfortunately not enough time to implement it. The idea was to take countries in similar situations during the pandemic and cluster them together based on mobility, epidemiology, restrictions, and more; allowing users to see how the clusters differ and develop over time.
- **Analyzing Tools:** One feature we wanted to add was giving the users a few tools to analyze the data further. For example, it would be nice to get a numeric value for the correlation on scatter plots.

- **Color Scale:** A feature to allow clients to choose the color scale would be an excellent addition to the application. The client could be allowed to select the rolling average described earlier and also some preset to quantize the scale. Quantizing the scale would allow the client to have an easier time seeing patterns on the map, but seeing the real difference between the two regions would be much more challenging. This feature could also add the previously discussed color scale using median and standard deviation, with the option to choose between all of them.

- **Automate BigQuery:** Automatically updating the reduced data sets by using BigQuery and Cloud Storage would ensure minimized data sets are always updated. Moving files from Cloud Storage to BigQuery and back is tedious, and would ideally be done automatically. A GitHub workflow could for example be used to run this process daily.

- **Bar Race for Multiple Regions:** The `BarRace` component currently only supports showing one region at a time. An improvement for this would be to allow for multiple regions to be shown at the same time. This could be done by taking the average values for each search trend and using it in the visualization.

- **Date Interval:** The pandemic has lasted for a long time, and data trends are likely to change over time. In addition, the application should allow users to look at a specific interval of dates. This feature was never implemented due to time constraints. Changing to a better-suited data structure for this, like a binary search tree, should be considered to enable this improvement.

- **Improve GeoJSON:** One of the Python scripts adds a location key column in the GeoJSON with the location key if it exists in the COD. If there is a mismatch in the codes used (ISO, FIPS, and NUTS) from Natural Earth, the application has no way of knowing about the data from COD. An improvement is to do the original idea discussed in 4.5.5 of finding location keys in the python script.

- **Testing:** During the development of the application tests were created to check that components mounted correctly. There was also some testing for specific events. These did not get updated after some of the updates to the data structures and are currently not working. The test suit should be improved upon.

- **Keep Data Loaded:** The hide data functionality is currently set up so that hidden locations are removed from the page state entirely. An improvement would be to keep this data in state in a separate variable, so the user does not have to reload this data when unhidden.

5.2 Conclusion

After the project, a web application was created, giving the user an overview of COVID-19 data by using different types of visualizations. It provides users with a straightforward interface that they can use to view the global situation of COVID-19. The first thing users see is the map, which provides an immediate overview of new cases worldwide. It is also possible to change what data is shown, such as new diseased, or vaccinations by using the sidebar. Users are able to select any country and/or its subregions to see data for the subregions. A preset of graphs are displayed under different tabs for different categories (epidemiology, vaccinations and search trends). It also allows users to dive deeper into detailed COVID-19 data using customizable graphs with multiple types of chart options, such as line charts and scatter plots.

In hindsight, more preparation should have been done to ensure the optimal method of slicing data was chosen before committing many hours to create python scripts for reducing the data. Looking back, it would have been wiser to allocate less time and effort to the compatibility issues that arose with the COVID-19 Open Data Repository and Natural Earth. The time spent on this was too much compared to the value of the results.

During the development of the application, we gained a broader knowledge of maps, and how to create them using Geo- and TopoJSON. We learned more about creating and hosting a website in addition to learning a lot about frameworks like React, TypeScript, D3.js, BigQuery, and Google Cloud Service, and how to use them effectively. We also gained a deeper understanding of how to use visualizations compellingly. This is the first project where we needed to properly consider handling data, as previous projects have not been at this scale, and a lot has been learned.

List of Figures

Listings

List of Tables

Appendix A

Instructions to Compile and Run System

Ensure node and npm has been installed in order to successfully run the application.

Clone the GitHub repository

```
$ git clone git@github.com:bachelor-group/bachelor-group.github.io.git
```

Change directory to `./bachelor-app`

```
$ cd bachelor-group.github.io/bachelor-app
```

Run `npm install` to install all dependencies and packages

```
$ npm install
```

Run the application

```
$ npm start
```

The application will open on localhost with port number 3000.

Appendix B

Benchmarking Results

B.1 concurrent version with 16 threads

```

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    36   909.059 MiB   909.059 MiB           1   @profile
    37                                             def generate_data(file, threads):
    38   909.059 MiB     0.000 MiB           1         urls = []
    39   944.918 MiB     0.000 MiB           2         with open(file, 'r') as csvfile:
    40   909.059 MiB     0.000 MiB           1             datareader = csv.reader(csvfile)
    41
    42                                                     # skip header
    43   909.059 MiB     0.000 MiB           1             next(datareader)
    44
    45   909.059 MiB     0.000 MiB         199             for row in datareader:
    46   909.059 MiB     0.000 MiB         198                 region_codes = row[0].split("_")
    47
    48   909.059 MiB     0.000 MiB         198                 url=""
    49   909.059 MiB     0.000 MiB         734                 for i,region in enumerate(region_codes):
    50   909.059 MiB     0.000 MiB         536                     url += region
    51   909.059 MiB     0.000 MiB         536                     if i != len(region_codes)-1:
    52   909.059 MiB     0.000 MiB         338                         url += "_"
    53   909.059 MiB     0.000 MiB         198                 urls.append(url)
    54
    55   909.059 MiB     0.000 MiB           1             try:
    56   944.918 MiB    30.219 MiB           2                 with concurrent.futures.ThreadPoolExecutor(max_workers=threads) as executor:
    57   914.863 MiB     5.641 MiB           1                     executor.map(write_to_file, urls)
    58                                                     except:
    59                                                         print()
```

```
Command being timed: "python ./Python/generate_data.py"
  User time (seconds): 24.39
  System time (seconds): 7.05
  Percent of CPU this job got: 84%
  Elapsed (wall clock) time (h:mm:ss or m:ss): 0:37.40
  Average shared text size (kbytes): 0
  Average unshared data size (kbytes): 0
  Average stack size (kbytes): 0
  Average total size (kbytes): 0
  Maximum resident set size (kbytes): 2229780
  Average resident set size (kbytes): 0
  Major (requiring I/O) page faults: 0
  Minor (reclaiming a frame) page faults: 803764
  Voluntary context switches: 310249
  Involuntary context switches: 6042
  Swaps: 0
  File system inputs: 0
  File system outputs: 5064
  Socket messages sent: 0
  Socket messages received: 0
  Signals delivered: 0
  Page size (bytes): 4096
  Exit status: 0
```

B.2 non-concurrent version with single thread

```

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    57  909.234 MiB  909.234 MiB           1   @profile
    58                                         def generate_data(file):
    59  916.309 MiB   -0.934 MiB           2       with open(file, 'r') as csvfile:
    60  909.234 MiB    0.000 MiB           1           datareader = csv.reader(csvfile)
    61
    62                                             # skip header
    63  909.234 MiB    0.000 MiB           1           next(datareader)
    64
    65  917.242 MiB -121.559 MiB         199           for row in datareader:
    66  917.242 MiB -122.090 MiB         198               region_codes = row[0].split("_")
    67
    68  917.242 MiB -122.090 MiB         198               url=""
    69  917.242 MiB -437.945 MiB         734               for i,region in enumerate(
 region_codes):
    70  917.242 MiB -320.371 MiB         536                   url += region
    71  917.242 MiB -320.371 MiB         536                   if i != len(region_codes)-1:
    72  917.242 MiB -198.281 MiB         338                       url += "_"
    73  917.242 MiB -122.090 MiB         198               try:
    74  917.242 MiB -115.016 MiB         198                   write_to_file(url)
    75  912.711 MiB   -1.465 MiB           2               except:
    76  912.711 MiB    0.000 MiB           2                   print()
```

```
Command being timed: "python ./Python/generate_data.py"
  User time (seconds): 31.15
  System time (seconds): 6.53
  Percent of CPU this job got: 36%
  Elapsed (wall clock) time (h:mm:ss or m:ss): 1:42.38
  Average shared text size (kbytes): 0
  Average unshared data size (kbytes): 0
  Average stack size (kbytes): 0
  Average total size (kbytes): 0
  Maximum resident set size (kbytes): 2229768
  Average resident set size (kbytes): 0
  Major (requiring I/O) page faults: 0
  Minor (reclaiming a frame) page faults: 825778
  Voluntary context switches: 26251
  Involuntary context switches: 2623
  Swaps: 0
  File system inputs: 0
  File system outputs: 5064
  Socket messages sent: 0
  Socket messages received: 0
  Signals delivered: 0
  Page size (bytes): 4096
  Exit status: 0
```

References

[1] Coronaviridae Study Group of the International Committee on Taxonomy of Viruses. The species severe acute respiratory syndrome-related coronavirus: classifying 2019-ncov and naming it sars-cov-2. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7095448/, 2022. [Online; accessed 14-May-2022].

[2] JON COHEN. Wuhan seafood market may not be source of novel virus spreading globally. https://www.science.org/content/article/wuhan-seafood-market-may-not-be-source-novel-virus-spreading-globally, 26 JAN 2020. [Online; accessed 13-March-2022].

[3] Axis Maps. Choropleth maps. https://www.axismaps.com/guide/choropleth, 2020. [Online; accessed 01-March-2022].

[4] D3js.org. Equirectangular. https://observablehq.com/@d3/equirectangular, 2022. [Online; accessed 14-May-2022].

[5] Nations Online. Country codes list. https://www.nationsonline.org/oneworld/country_code_list.htm, 2022. [Online; accessed 09-May-2022].

[6] Destatis Statistisches Bundesamt. Nuts classification. https://www.destatis.de/Europa/EN/Methods/Classifications/OverviewClassification_NUTS.html, 2022. [Online; accessed 13-May-2022].

[7] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, pages 238–255, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03237-0.

[8] TypeScript Documentation. TypeScript for JavaScript Programmers. https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html.

[9] Rachel Appel. Write Object-Oriented JavaScript with TypeScript. https://rachelappel.com/2015/01/02/write-object-oriented-javascript-with-typescript/.

[10] Stack Overflow Developer Survey. Web frameworks. https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks/.

[11] Web Dev Zone. DOM Manipulation in React. https://dzone.com/articles/dom-manipulation-in-react.

[12] React Documentation. Components and props. https://reactjs.org/docs/components-and-props.html, 2022. [Online; accessed 2-April-2022].

[13] React Documentation. Introducing jsx. https://reactjs.org/docs/introducing-jsx.html, 2022. [Online; accessed 2-April-2022].

[14] Dave Ceddia. React state management libraries and how to choose. https://daveceddia.com/react-state-management/, March 17, 2021. [Online; accessed 2-April-2022].

[15] Mike Bostock. D3.js - data-driven documents. https://d3js.org/, 2021. [Online; accessed 01-March-2022].

[16] Evan Li. Github ranking. https://github.com/EvanLi/Github-Ranking#javascript, 2022. [Online; accessed 22-March-2022].

[17] Mike Bostock. D3.js - data-driven documents. https://d3js.org/#selections, 2021. [Online; accessed 22-March-2022].

[18] World Wide Web Consortium (W3C). Selectors api level 1. https://www.w3.org/TR/selectors-api/, 2021. [Online; accessed 22-March-2022].

[19] Yan Holtz. The d3 graph gallery. https://d3-graph-gallery.com/, 2018. [Online; accessed 22-March-2022].

[20] World Wide Web Consortium (W3C). Scalable vector graphics (svg) 2. https://www.w3.org/TR/SVG2/, 2018. [Online; accessed 22-March-2022].

[21] Wikipedia. Google cloud platform. https://en.wikipedia.org/wiki/Google_Cloud_Platform, March 15, 2022. [Online; accessed 13-April-2022].

[22] Google. Bigquery. https://cloud.google.com/bigquery, 2022. [Online; accessed 23-March-2022].

[23] Google Cloud. Key terms. https://cloud.google.com/storage/docs/key-terms, April 12, 2022. [Online; accessed 13-April-2022].

[24] Natural Earth. Natural earth. https://www.naturalearthdata.com/, 2022. [Online; accessed 22-March-2022].

[25] Esri. Shapefile file extensions. https://desktop.arcgis.com/en/arcmap/latest/manage-data/shapefiles/shapefile-file-extensions.htm, 2022. [Online; accessed 23-March-2022].

[26] json.org. Introducing json. https://www.json.org/json-en.html. [Online; accessed 23-March-2022].

[27] H. Butler et al. Rfc 7946 - the geojson format. https://datatracker.ietf.org/doc/html/rfc7946#section-1.5, August 2016. [Online; accessed 23-March-2022].

[28] Mike Bostock. How to infer topology. https://bost.ocks.org/mike/topology/, September 2, 2013. [Online; accessed 24-March-2022].

[29] O. Wahltinez et al. Covid-19 open-data: curating a fine-grained, global-scale data repository for sars-cov-2. 2020. URL https://goo.gle/covid-19-open-data. Work in progress.

[30] d3. d3 scale chromatic. https://github.com/d3/d3-scale-chromatic, 2018. [Online; accessed 10-May-2022].

[31] Google Docs. Covid-19 search trends symptoms dataset. https://github.com/GoogleCloudPlatform/covid-19-open-data/blob/main/docs/table-search-trends.md, 2022. [Online; accessed 29-April-2022].

[32] Matt Hinchliffe. React tag autocomplete. https://github.com/i-like-robots/react-tags, 2022. [Online; accessed 24-April-2022].

[33] Google. gsutil tool. https://cloud.google.com/storage/docs/gsutil. [Online; accessed 26-April-2022].

[34] GitHub. About github pages. https://docs.github.com/en/pages/getting-started-with-github-pages/about-github-pages#usage-limits, 2022. [Online; accessed 08-May-2022].

[35] GitHub. About github pages. https://docs.github.com/en/pages/getting-started-with-github-pages/about-github-pages#usage-limits, 2022. [Online; accessed 08-May-2022].

[36] Miguel. Which one is best csv or json in order to import big data (php). https://stackoverflow.com/questions/26156646/which-one-is-best-csv-or-json-in-order-to-import-big-data-php, 2020. [Online; accessed 23-April-2022].

[37] Maksym Goroshkevych. When does it make sense to use google bigquery. https://daveceddia.com/react-state-management/, 2021. [Online; accessed 21-April-2022].

[38] Google. Transcoding of gzip-compressed files. https://cloud.google.com/storage/docs/transcoding, 2022. [Online; accessed 23-April-2022].

[39] Esteban Borges. Gzip vs zip: difference between the most popular compressing file formats. https://nixcp.com/gzip-vs-zip-differences/, November 17, 2017. [Online; accessed 23-April-2022].

[40] StatCounter. Search engine market share worldwide. https://gs.statcounter.com/search-engine-market-share, 2022. [Online; accessed 13-March-2022].

[41] Chessrat at English Wikipedia. https://commons.wikimedia.org/w/index.php?curid=50060468. [Online; accessed 29-April-2022].

[42] Axis Maps. Map projections. https://www.axismaps.com/guide/map-projections, 2020. [Online; accessed 01-March-2022].

[43] Intergovernmental Committee on Surveying and Mapping. Commonly used map projections. https://www.icsm.gov.au/education/fundamentals-mapping/projections/commonly-used-map-projections, 2022. [Online; accessed 01-March-2022].

[44] PROJ. Equidistant cylindrical (plate carrée). https://proj.org/operations/projections/eqc.html, 17 Feb 2022. [Online; accessed 01-March-2022].