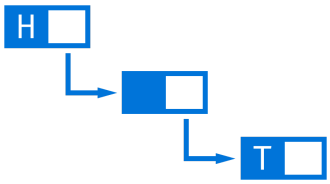
Final Report



Team: Linked List

Member	zID	Role
Thomas Sinn	z5213546	Project organisation, Scrum master, frontend developer
Hayes Choy	z5258816	Architecture, Backend Lead
Eddy Wong	z5207607	Overall Documentation, backend developer
Leila Yuan	z5261559	API, backend developer, Stoplight documentation
Xiyang Shi	z5137765	API, Frontend Lead

Please refer to table of contents, as this report includes prior reports and therefore is a very lengthy document

0 Table of Contents

Details

Name	Account Manager	Link
Google Drive	Thomas	https://drive.google.com/drive/folders/1ltCDccVZqfn5X3XfyBA7COY-HtRUY9Ou
GitHub	Hayes	https://github.com/hayeselnut/SENG3011_Linked_List
Trello	Thomas	https://trello.com/b/f0MnxHBc/seng3011
StopLight	Eddy	https://seng3011.stoplight.io/
Google Cloud Functions / Firebase	Leila	https://console.firebase.google.com/u/6/project/still-resource-306306/overview/
Discord	Thomas	https://discord.gg/c7vmj4Rc

1 Deliverable 4 (Final summary)

1.1 User Requirements & Use Cases

1.1.1 Phase 1: Linked-List API

Git hub Repo	https://github.com/hayeselnut/SENG3011_Linked_List
API	https://australia-southeast1-linked-list2.cloudfunctions.net/api
Stoplight	https://seng3011.stoplight.io/docs/seng3011-linked-list/reference/CDC.v1.yaml

1.1.1.1 User Requirements

The requirements of the Linked-List API are stated in detail in the project specification. In phase 1, we implemented each component specified in the project requirement while also adding complementary features which serves the purpose of a better user experience and aiding us in our analytics platform in phase 2.

The requirements from the project specification include:

- Requirement 1: Develop a scraper to gather data from the main data source.
    - ***Functionality***: We have developed a scraper that is able to scrape hundreds of pages on CDC.
    - ***Data content/format***: The scraped data contains all required information and follows the JSON format given in the project specification. In addition to that, we give unique IDs to all objects, including articles, reports and locations.
    - ***Data store***: The JSON data is then fed into a Firestore database.
    - ***Deployment***: Since the runtime of the scraper is over 9 minutes, we only partially deployed it on Google Cloud. To completely run the scraper, we manually operate it offline.
    - ***Miscellaneous***: In order to assist in the performance for our business product, we developed an extra scraper for https://www.api.org/news-policy-and-issues/pandemic-information/state-pandemic-resources so that we have a sufficient number of medical reports for states in the US.

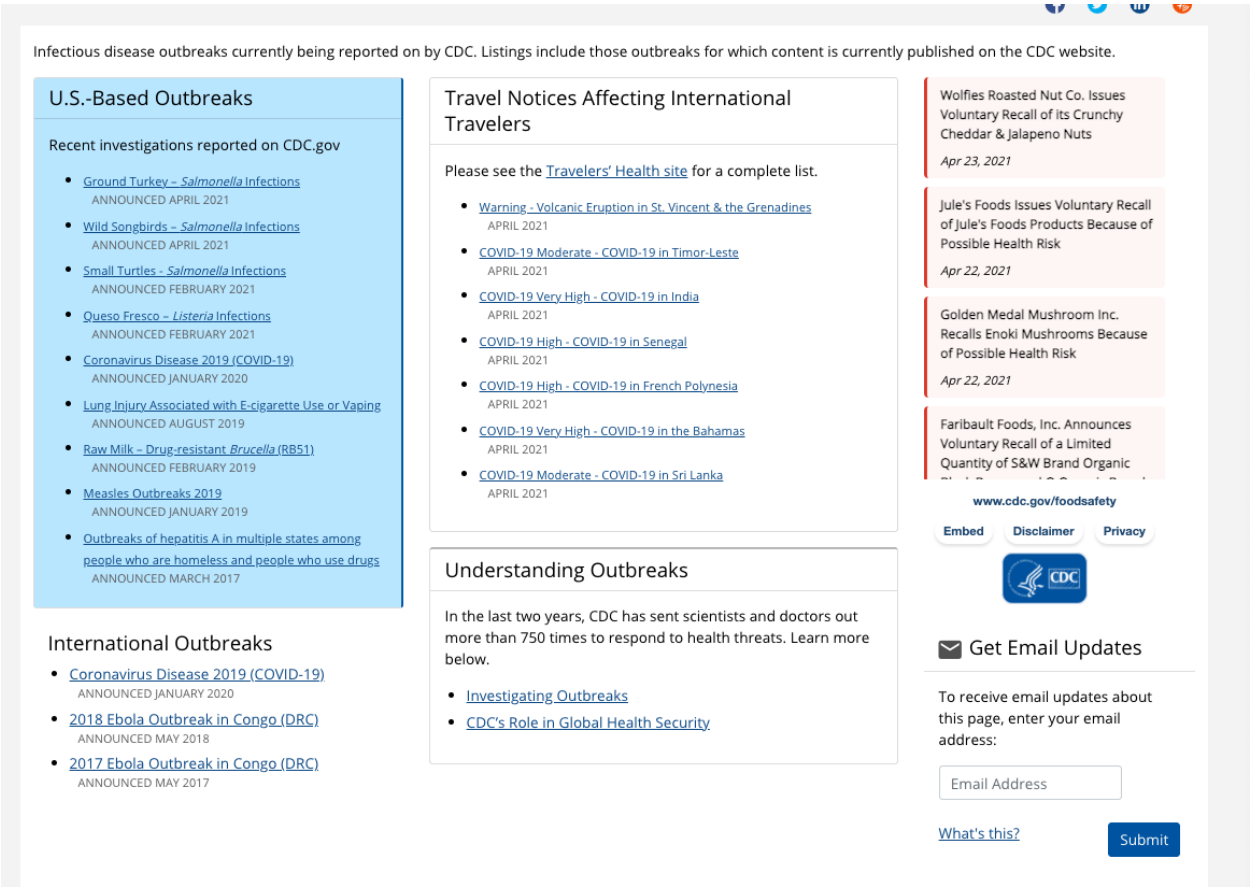
- Requirement 2: Develop an API to provide disease reports on demand, following the input and output formats defined.
    - ***Functionality***: We developed an API which follows REST design principals in order to enable users to interact with the database and return required information based on user's queries. The API we developed can be called from the API link provided above.
    - ***Required Usage***: The API we developed enables clients to find and access all the disease reports related to a given search term including disease, and location for a given period of time. These will be further addressed in use cases.
    - ***Additional Usage***: Besides the basic search terms mentioned above, we also allow users

to access articles through article IDs. With those search terms, users can further perform negative/inverse/exclusion matching. These will be further addressed in use cases.

- ***Deployment***: It was deployed via google cloud functions and grabs data from our firestore database.
- ***Documentation***: Our API is documented with spotlight. Developers are able to test and interact with our API through spotlight documentation. The link to our documentation is added above.
- ***Testing***: We have performed unit tests, speed tests, security tests and pressure tests. Unit tests were used to test on individual functionality of the API, which we managed to pass all of them. The speed tests were used to test the time for the API to respond, which in our case reflected the inefficiency of our firestore as it was located in the US (default option) instead of Australia. Pressure tests checked the number of users our API was able to handle at the same time, whereas security tests were based on the pressure tests to address the DDox attack.

1.1.1.2 Use Cases

- As a user, I want the data I query for coming from a reliable source so that I know the data is valid.
    - The data we used was from the CDC which is a trusted and reliable source



- As a user, I want to be able to query for medical articles/reports by giving a time period so that I can collect reports within my period of interest.

    - This is satisfied by the API and can be viewed via the API documentation here
    - https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-02+13%3A00%3A00+to+2021-04-29+00%3A00%3A00

- As a user, I want to be able to query for medical articles/reports by giving a location so that the reports I collect are from the area I am interested in.
    - For location name matching, we accept multiple expressions (eg. US, United States, United State etc are treated the same)
    - This is satisfied by the API and can be viewed via the API documentation here.
    - https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-02+13%3A00%3A00+to+2021-04-29+00%3A00%3A00&location=US
    - https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-02+13%3A00%3A00+to+2021-04-29+00%3A00%3A00&location=United+States
    - https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-02+13%3A00%3A00+to+2021-04-29+00%3A00%3A00&location=United+State

- As a user, I want to be able to query for medical articles/reports by specifying a key term/multiple key terms so that the reports I collect are associated with the key termI am interested in.
    - For key term matching, we accept multiple expressions (eg. COVID-19, corona etc are treated the same)
    - This is satisfied by the API and can be viewed via the API documentation here
    - https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-02+13%3A00%3A00+to+2021-04-29+00%3A00%3A00&key_terms=corona
    - https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-02+13%3A00%3A00+to+2021-04-29+00%3A00%3A00&key_terms=COVID-19
    - https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-02+13%3A00%3A00+to+2021-04-29+00%3A00%3A00&key_terms=fever

- As a user, I want the medical reports/articles I query contain disease names, symptoms, event date and locations so that I can have a basic understanding on .
    - This is satisfied by the API and can be viewed via the API documentation here
    - https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-02+13%3A00%3A00+to+2021-04-29+00%3A00%3A00&key_terms=salmonellosis

- As a user, I want to be able to perform negative/inverse/exclusion matching on the query so that I am able to address all my requirements in one query.
    - This is satisfied by the API and can be viewed via the API documentation here
    - https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-02+13%3A00%3A00+to+2021-04-29+00%3A00%3A00&key_terms=Corona%2C+%21Ebola&location=United+States
    - https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-02+13%3A00%3A00+to+2021-04-29+00%3A00%3A00&key_terms=fever%2C+%21salmonellosis&location=United+States

- All the use cases are satisfied via the single API call with the use of different arguments.

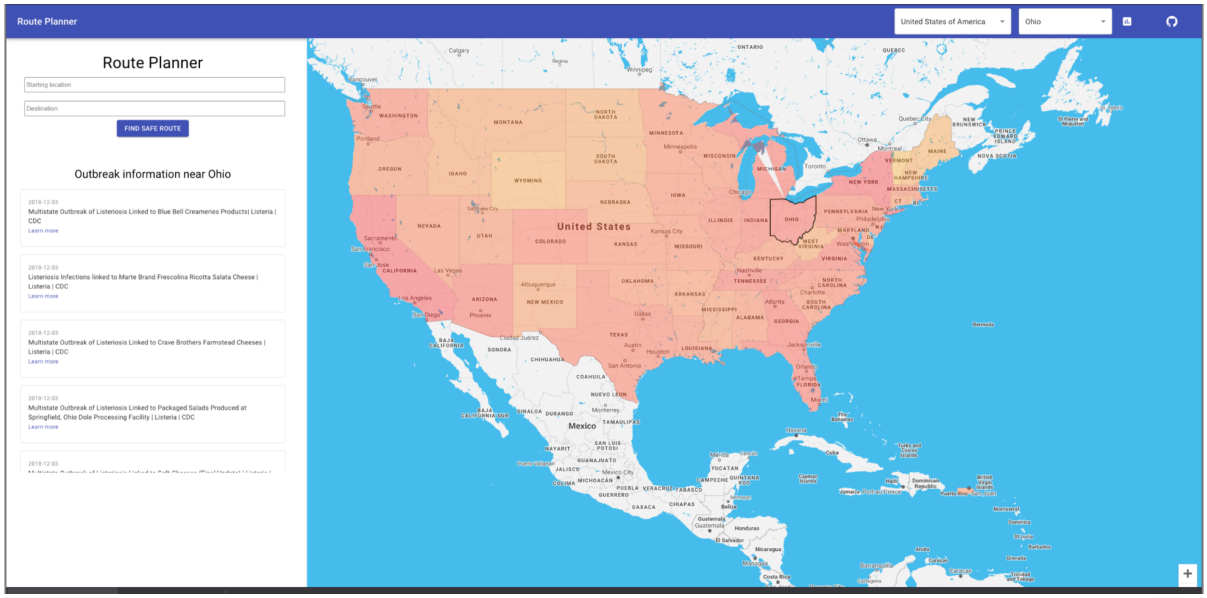
1.1.2 Phase 2: Analytics platform

Git Hub Repo	https://github.com/hayeselnut/SENG3011_Linked_List/tree/main/PHASE_2
How to start	1. Follow this: a. PHASE_2/Application_SourceCode/app b. Run yarn install c. Run yarn start d. Open http://localhost:3000/map in browser, if it is not auto opened

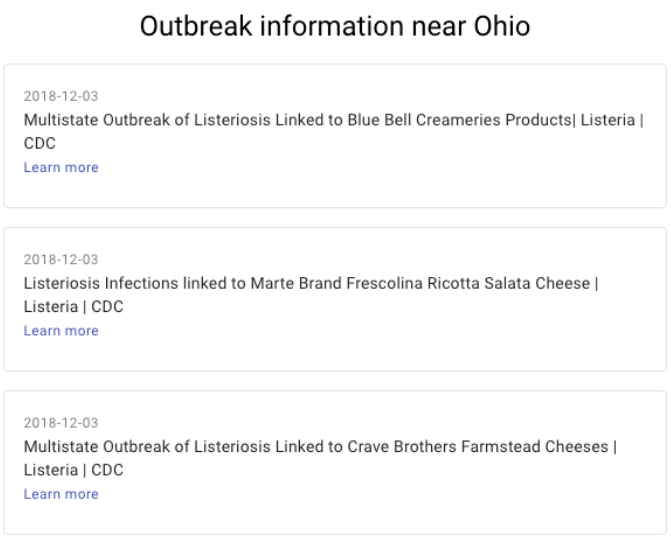
1.1.2.1 User Requirements

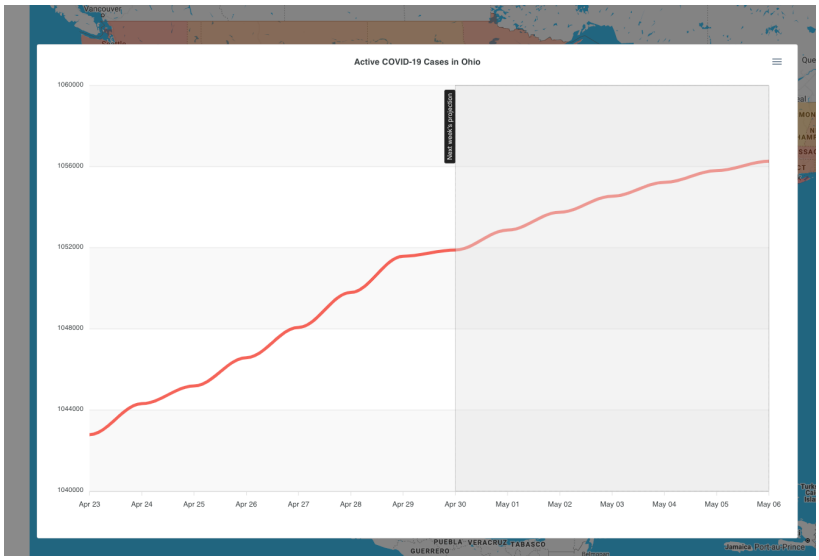
For phase 2 the following requirements include:

- Developing a web application that integrates disease reports from the API you developed and other APIs.
    - This is evident in our react app which can only be deployed locally, via the how to start guide above.
    - Here is a screenshot for reference



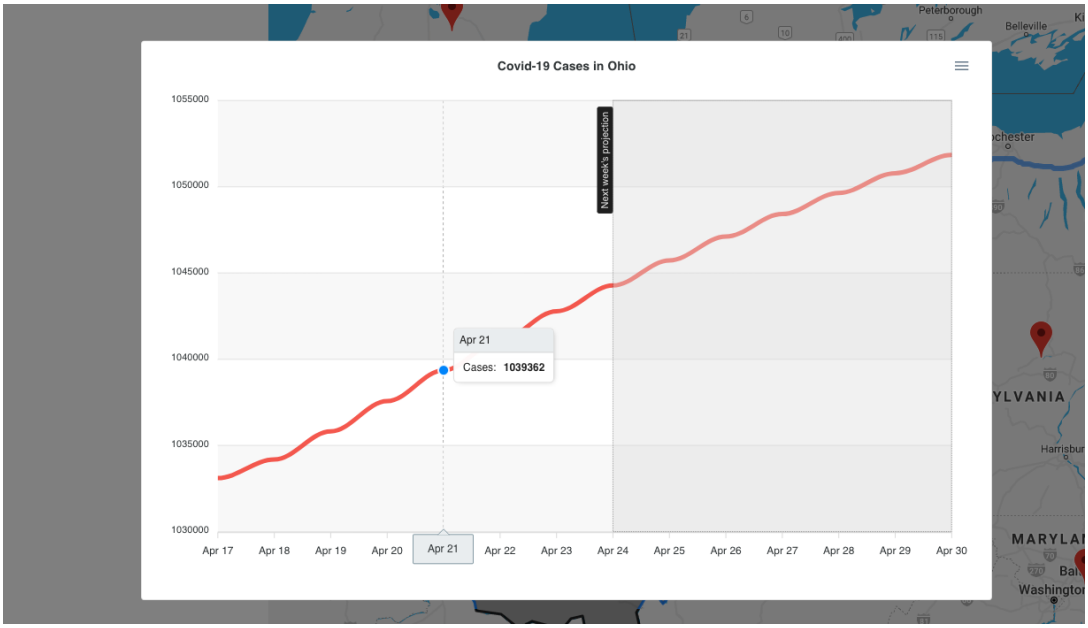
- Using the interface, users are able to view articles or reports relating to the state selected while also being able to visualize reports over time, as can be seen by the article cards on the left and the graph to the right, where the brighter region shows covid cases over time.



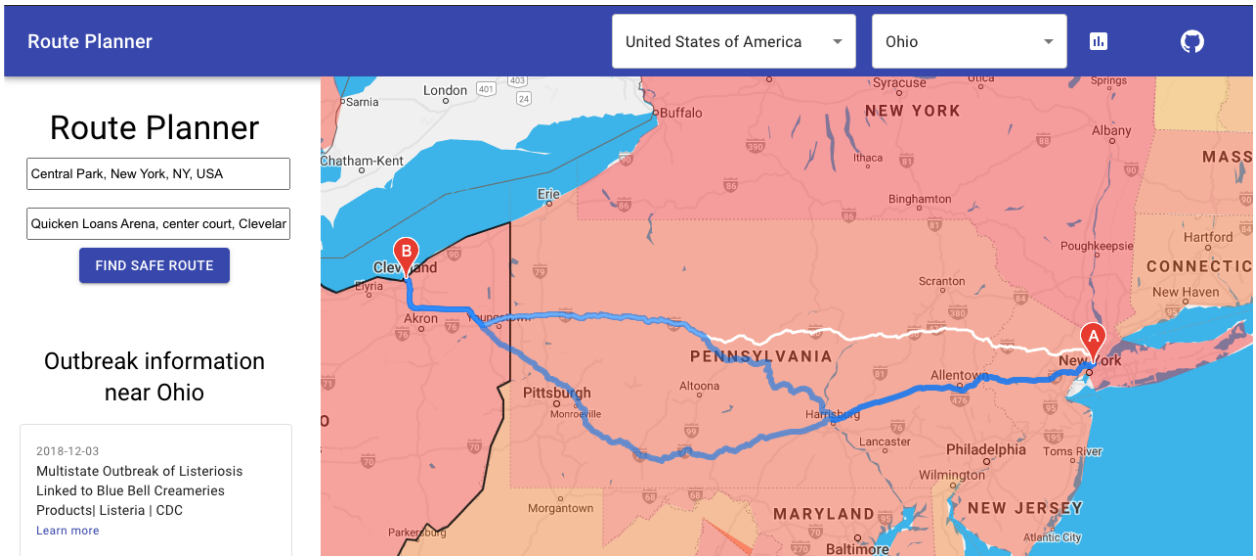


In addition to these features, there were some optional requirements which we have also decided to fulfil

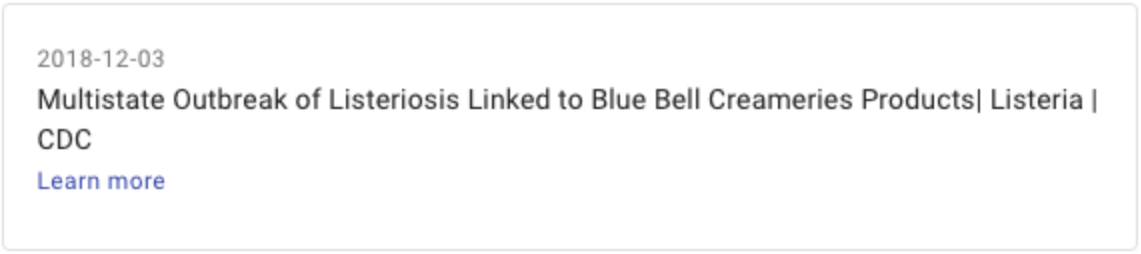
- ***Prediction based features which enable us to predict things in the future***
  - In our app we utilized a neural network using LSTMTimeStep template in the brain.js.
  - Furthermore hovering over a certain point of time on the graph, displays the number of recorded covid cases in that state.



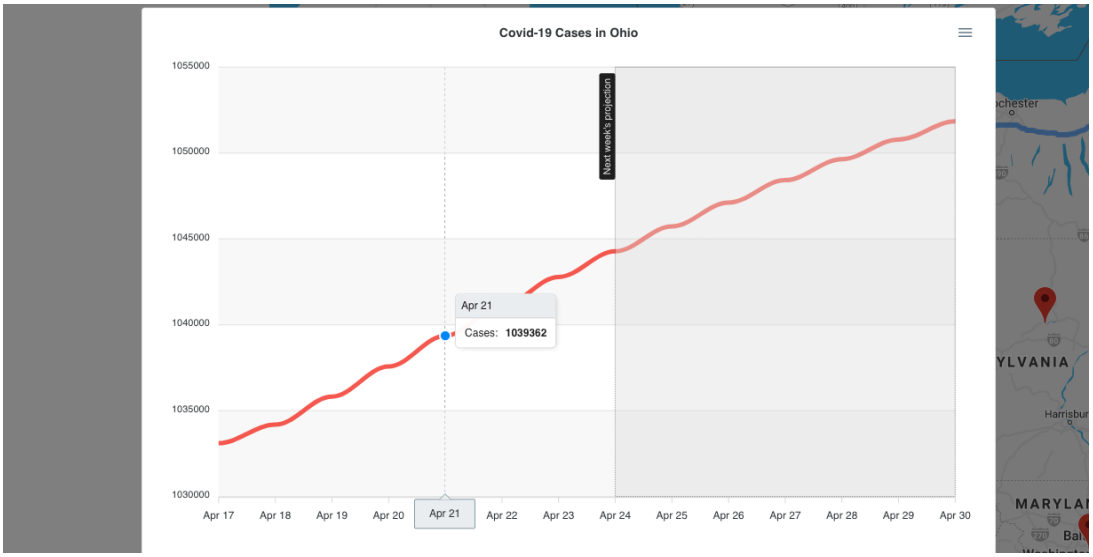
- ***Enhancing the overall data visualization and the user experience in navigating thru large numbers of disease reports***
    - In our app we provided a dashboard like view to enable the user to have an overview of their route plan, while also having access to disease articles, reports and predictions, this can be seen below with the route outlined between Central park new york and Quickens Loans Arena



    - This can be seen in the screenshot below, which shows the user has access to
        - **Articles** via the bottom left corner, which are linked to the article itself and can be redirected to via the '*Learn more'* hyperlink



- **Reports and Covid outbreak numbers are accessible via this icon** on the nav bar at the top, which will show you the resulting graph

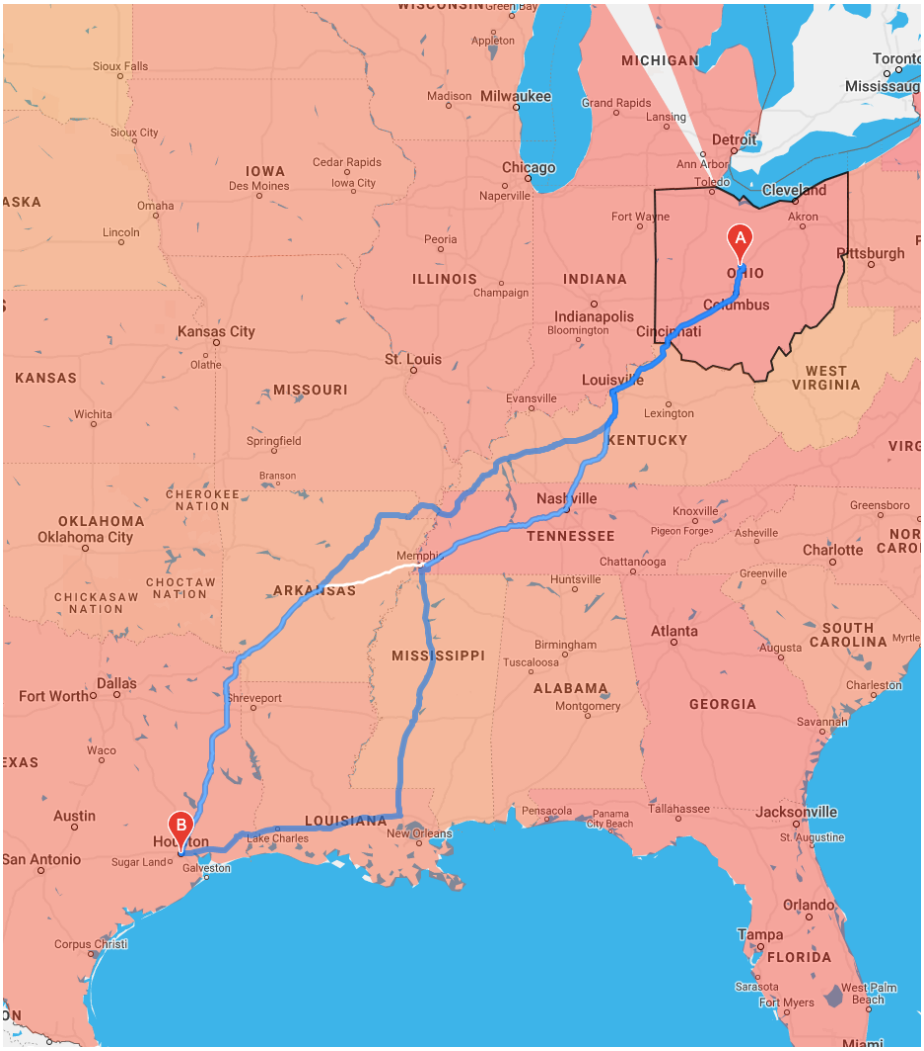


- **Predictions** via the graph also which is indicated via the darker region in the graph.

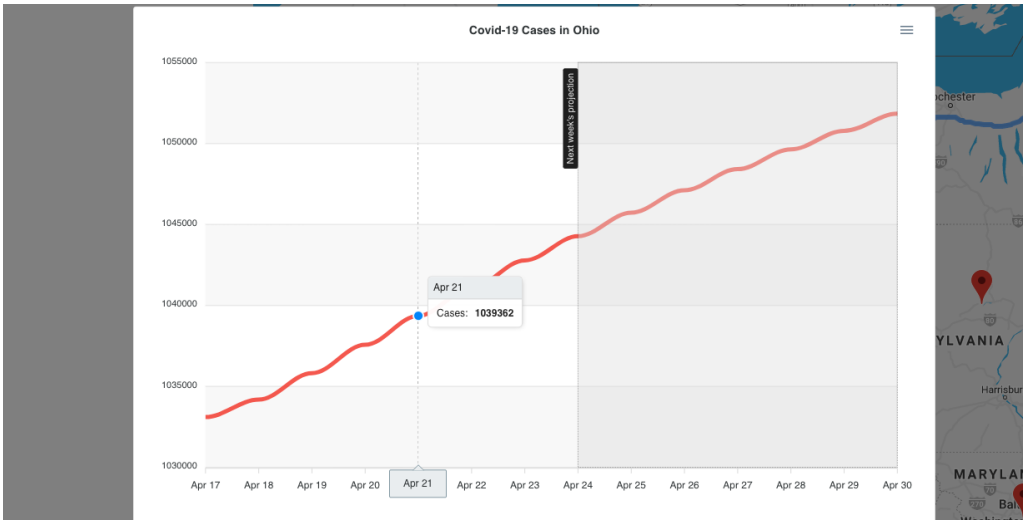
1.1.2.2 Use Cases

- As a user I would like a way to input states in the dashboard, so that the route planner knows which states I would like to travel.
  - This feature is seen through the use of two drop down menus and a search box.
  - The use of the drop down menu is to ensure ease of use for the user, due to limitations of the solution
  - While the use of a search box is to enable the user to easily search cities

- As a user I want to be able to provide multiple routes along with the COVID-19 situation for the cities each route passes through so that I will be able to choose a relatively safer route.
    - This feature is seen in the automatic generation of a route from point a to point b when a state is entered in the drop down menu and search box
    - This can be seen by the three routes, there are 2 in blue, and 1 in white under the blue route
    - These routes were determined on a city basis depending on the severity of the covid situation
    - Which was determined by creating an index, which was weighted 9/10ths by covid stats and 1/10th on articles scraped by our two scrapers



- As a user I want to see a forecast of future covid cases, so that I can plan around those states/cities in my future(1 week in advance at most) roadtrip
    - This feature is evident when you click on the graph icon in the nav bar



- As a user I want to see a heat map of all active covid cases, so that I can plan around those states/cities
    - This is evident in our map screenshot that can be seen below with the varying shades of green - red which indicate the severity of the active covid cases
    - Green being the safest
    - Red being the most dangerous

1.2 System Design and Implementation

- Most if not all feedback has been addressed

1.2.1 Software Architecture Overview & API's

1.2.1.1 Phase 1: Linked-List API

The API final design and implementation didn't change, to jump to API design click here: **3 Deliverable 2 (API design)**

1.2.1.2 Phase 2: Web Analytics Platform

The front end design is laid out in d3, although we went through some implementations in deliverable 1, we fleshed out most of the app by d3. For more information click here: **2 Deliverable 3 (APP Design)**

1.2.1.3 Key Benefits and Achievements

Phase 1 - API

The key benefits and achievements of this project are purely subjective. As a team in SENG3011 we initially achieved the same result as everyone else, producing a scraper that can gather data from a datasource. For us this was a big achievement overall as we as a team learnt many things, these include the use of google cloud products, like firestore and cloud functions and API documentation tools like stoplight. These two main tools enabled us to deploy an online serverless API which uses a scraper to gather data from the CDC website. The benefits of such an API enables users to simply use the API endpoint in a browser or terminal to query data from our database in a fast and reliable manner due to our cloud deployment on google cloud platform, which can scale up or down depending on the usage of our API.

Phase 2 - Web App

The project outlines a webapp that visualizes or makes predictions based on any disease data. For our web app we decided to create a covid route planner which provides a centralized dashboard for users to have up to date articles on related diseases in the area, graphical representation and predictions made from past covid data and a route planner which navigates around covid hotspots, with directions that can be exported for use in other apps eg. google maps. The key benefits can be seen as the positive value we have right now, which includes providing a dashboard for covid related data and a covid route planner, which benefits individuals who plan to make road trips in the near future, furthermore the data is accurate to 1 - 3 days.

1.3 Team organisation and conclusion/appraisal of your work

1.3.1 Responsibilities/organization of the team

Phase 1 - API

Name	Responsibilities
Thomas	● Scrum Master ● Minutes taker ● API documentation manager ● Secondary tester
Hayes	● API end point developer ● Back end developer (node.js)
Bob	● Primary tester ● Testing report writer
Leila	● Back end developer (python) ● Primary scraper developer ● Cloud firestore manager
Eddy	● Back end developer (python) ● Secondary scraper designer ● Cloud deployment manager

Phase 2 - Web App

Name	Responsibilities
Thomas	● Scrum Master ● Minutes taker ● Front end developer (javascript, react)
Hayes	● Front end developer (javascript, react)
Bob	● Front end developer (javascript, react)
Leila	● Overall Documentation ● Script writer
Eddy	● Overall Documentation ● Script writer ● Report Writer

1.3.2 Major achievements in project/problems encountered

Overview

Quite often our biggest achievements were the solutions to issues and/or problems encountered, hence the combination of both headings, furthermore each deliverable has been updated to reflect the current state of our project.

Phase 1

For phase 1 the biggest achievements were briefly explained, however to further reiterate the point, the major achievements overall was the development of the scraper, API and using cloud platform to deploy all this.

Phase 1 - Scraper

The scraper itself was a massive achievement as we knew the basics of how to make a scraper, however never developed a scraper that performed such in depth analysis on it's scraped content. For example to gather specific information on diseases we used a string matching library called fuzzy wuzzy which would match on the exact disease name or other common names for that disease, this was powerful as it enabled us to gather disease information. Furthermore we used beautiful soup, which enabled us to match on tags in a html page, datefinder to match on dates which we used for reports and geograpy to match on any and all cities, states and countries. Thus overall a very big achievement for us.

Phase 1 - Deployment

Another achievement was the deployment of our API on Google's cloud services. As a team we had little to no experience on this platform, however as a team we collectively learnt the uses and found it a very rewarding experience. As we were able to deploy our scraper, database, documentation and API and have it solely operate on a cloud basis.

Phase 2

In phase 2 the biggest achievement by far is implementing our idea, as the idea of a route planner daunted us, due to the many components of data we had to deal with such as accurate coordinates and reliable covid statistics on each city and state.

Phase 2 - Maps

For maps the biggest achievement and challenge we had was finding a solution to produce a border around a state, which served as means to highlight the state we selected as well as in use for our heatmap, as we had a file with the longitude and latitude points of a state's border, the most difficult point was making a polygon that would be the same if not similar shape of a state, we struggled on this for a while as the polygon we produced never matched the state shape, until we stumbled upon a google maps API which had a function polygon which enabled us to match the shape of state. Thus being one of our biggest challenges however a big achievement as it provides an immensely satisfying user experience being able to see a heat map and border around the selected states.

Phase 2 - Route Planning

For maps hands down the best achievement we achieved was being able to produce a route when given two points (origin to destination) as it is the main business value of our app, this was done via the use geocoding API which from a street address returns a longitude and latitude point. This is then fed into the directions API which provides us with a JSON with directions and this is finally pushed to the front end using a function called DirectionsRenderer from Google, which creates a line that matches the directions needed for the route.

Phase 2 - Predictions

Lastly the final issue we encountered that we didn't solve due to not being feasible, was the use of machine learning, which was not feasible as each state required a different configuration which resulted in very long start times and thus hinder the user's experience. For more details refer to **2 Deliverable 3 (App Design)**

1.3.3 What skills you wish you had before the workshop

Overall the workshop was a challenging and rewarding experience, as the point of these software courses is to enrich our problem solving as well as providing group context to better prepare us for the future. There are a few skills we wish we had before this, they include a proficiency in:

- React JS & Javascript
- Cloud Computing
- Natural Language Processing
- Web scraping

For React and Javascript, this was a big part of our project, being that it was the whole of Phase 2, and that only ⅗ team members were proficient in this. As a result this led to many long nights for the front end guys, where others couldn't help. Furthermore if more of us knew more it would've led to a better front end! Thus to conclude the skills we wished we had would be a better foundational understanding of React JS and Javascript.

Furthermore the concept of cloud computing was foreign to a few team members as we've never heard of this. It was very challenging to think of software as a service, as we'd never used external third party solutions as extensively as this. Most courses tend to push you to develop your own solutions from the ground up, which has the negative implication of less development time and thus less time implementing the business solution. It is a skill that some of us wish we had, as the lack of understanding of this led to deployment of our scraper being half fast. However it is a skill that we found invaluable and loved learning about!

Natural Language processing and web scraping go hand in hand, they are skills that were partly taught as a part of COMP2041, however as a team we feel that it should've been fleshed out more (only was one week in comp2041), being that they are valuable skills used when gathering important stats. Thus being

able to develop those skills in a lecture-like environment would result in better quality web scraper solutions, which ultimately provides a better user experience!

1.3.4 Would you do it any differently now ?

The things that we wish we did differently are all skills we wish we had more developed before attempting this course. These include more experience in cloud computing and more developed front end human resources to then be able to showcase a better product/solution, as the point of these courses is to emulate what the software engineering world is like! Majority of these concerns are addressed in the skills we wish we had before the workshop.

2 Deliverable 3 (App Design)

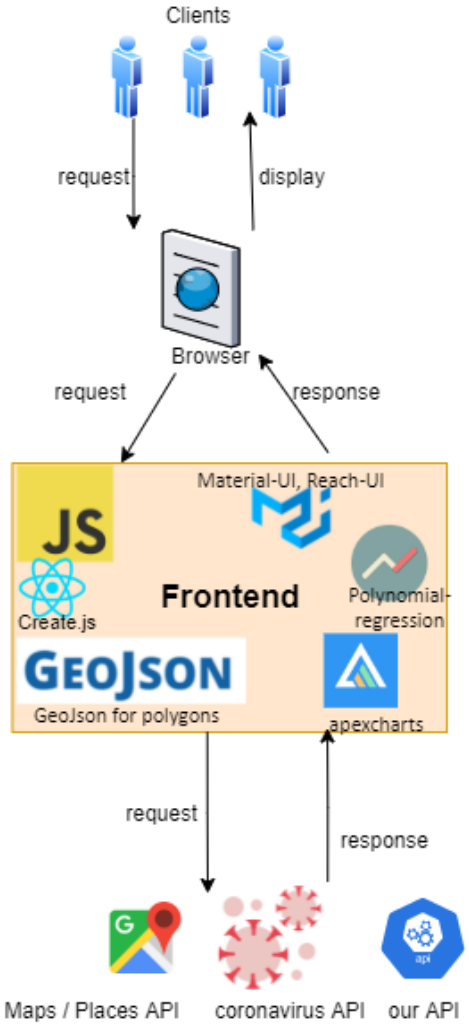
2.1 Development environment

In phase 2 we developed a platform that acts as a route planner, which is able to provide users feasible and safe routes for interstate travels in the U.S and India. The frontend was developed in ReactJS in combination with Material UI's extensive component library because it is easy to prototype quickly as well as provides a unified aesthetic across the application. In addition to this, since Google Maps API has a npm library it was very easy to integrate an interactive map and predictive search with this development environment.

2.2 System Architecture

- To the right is a brief system architecture of how front end is built.
- More is explained in detail below
- When users interact with the React UI, requests are sent and handled by JS which then initiates the given feature.

- We used:
    - ***React JS and JS*** to develop the map and the view you see
    - ***Material - UI and React UI*** to grab easy and quick to use components to create an aesthetically pleasing and satisfying UI
    - ***Create JS*** is another component library we made use of
    - ***JS-Polynomial Regression*** is a javascript library that provides us with predictions based on a polynomial regression formulae
    - ***GADM(GeoJson Date) and Google Maps APIs*** are used to create polygons and provide map like functionality
    - ***Apex charts*** is used to create the graphs that are seen below



How a route is determined

1. First two points are selected via the text boxes.
    a. As the user types, locations are autofilled and matched using the Google Places API which enables the user to grab specific location names for their route.
    b. These two points are then inputted into the Google Geocoding API which returns a long and lat coordinate based on street address
2. These two coordinates are then sent to Google Directions API which returns JSON of the route
    a. The great thing with using Google APIs, the formatting is consistent and easy to pass between APIs
    b. The JSON route is then modified to better reflect detours around active covid cases
    c. The Covid-19 data is acquired from a
3. Then sent to Google Directions Renderer which when given a map and a route, it maps the route on the map.

How a predictive graph is determined

1. Data is grabbed from the Covid-19 API
2. The data points are then run through the polynomial regression formula
3. The data points are then inputted in the Apex charts API which produces an interactive graph that we can embed in the React UI

How a heat map is determined

1. Data is grabbed from the Covid-19 API on a state level
2. The data is then indexed from most severe to least severe
3. Colored accordingly in our algorithms and then using the polygon functions we filled the polygon and pushed it to front end.

2.3 API usage/external data source

The list of apis we interacted with are as follows!

- Covid-19 API
    - **Documentation:**https://documenter.getpostman.com/view/10808728/SzS8rjbc#81447902-b68a-4e79-9df9-1b371905e9fa
    - Provides the number of current/past active cases for cities in the U.S and India
- Map API
    - https://cloud.google.com/maps-platform/maps
    - Provides an interactive map displaying all states in the U.S.and India
    - Provides multiple routes and displays them on the map by giving a starting point and a destination, along with the coordinates associated with the routes.

- Places API
    - https://cloud.google.com/maps-platform/places
    - Provides predictive search functionality to allow users to easily find the places that they are going to be travelling to.
    - Provides the location information (country, state, city etc) from a set of coordinates. This is particularly used in getting the city names along a route. Those city names allowed us to find out the number of active cases for them hence analyse how safe a route is.
- GeoCoding API
    - https://developers.google.com/maps/documentation/geocoding/overview
    - Is a Google API which takes in a street address and converts it into geographic coordinates like (longitude and latitude), we used this particularly to produce the markers you see on the map as well using them in creating a route.
- Route API
    - https://cloud.google.com/maps-platform/routes
    - A Google API which is heavily utilized in determining the route as it returns a JSON with directions from point a to point b, with the use of waypoints, which can then be used in our functions to create a route on the map.
- Linked_List API
    - https://seng3011.stoplight.io/
    - Provides related medical reports from CDC (https://www.cdc.gov/outbreaks/) and American Petroleum Institute (https://www.API.org/news-policy-and-issues/pandemic-information/state-pandemic-resources) for each state as further reading materials for users to explore
    - One top of CDC, in this phase we scrapped one more site hosted by the American Petroleum Institute, which allowed us to have more reports for each state in the U.S regarding COVID-19.

Modules/tools usage

We have used several modules to assist us in frontend development which are detailed below.

- Material-UI
    - https://material-ui.com/
    - We used material-UI to ensure we deliver a visually-appealing application. With different components including Container, Grid, Paper, Typography, Link, Button etc allowed us to create a user-friendly and unified interface in React. Using someone else's component library means that we can prototype and build in a faster and more agile manner.

- Reach-UI
    - https://reach.tech/
    - Same as Material-UI, we have also used Reach-UI, a Reactor-based design system to reinforce the aesthetic of our interface. We mainly used its combobox design to structure the interface. (eg. enter the state combobox)
- Js-polynomial-regression
    - https://www.npmjs.com/package/js-polynomial-regression
    - Our application provides predictions about the active cases in the future for each state in the U.S so that users can forecast their future road trip plan. We achieved this by implementing a polynomial regression model. The model takes the number of active cases for the past 7 days in a state and fits it into a polynomial regression model, which enables us to predict the number of active cases for the next 7 days.
- React-apexcharts
    - https://apexcharts.com/docs/installation/
    - We integrated Apexcharts into our react.js application to create a linear chart that is able to show the prediction of COVID-19 cases in the future (introduced above). By inputting components including title, grid, annotations etc we were able to present the linear graph clearly and aesthetically with a line separating past data and predicted data.
- GeoJson for state polygons
    - https://gadm.org/maps.html
    - In order to be able to outline a state, we used the dataset provided by GeoJson. The Geodata package it provides allowed us with geojson polygons for the U.S states. We saved the required coordinates as local js files and used them to plot the outline of a selected state.

Implemented Algorithms

We implemented the polynomial regression algorithm in order to make predictions about the number of active cases for the next 7 days based on the data we have for the past 7 days.

For making predictions, we have considered another approach and spent a long time trying to achieve it. Unfortunately we failed to implement it fully and went back to our original approach due to a series of difficulties.

For a more advanced approach, we tried to use the neural network library with the LSTMTimeStep template in the brain.js (long short term memory) which was recommended for predicting data. We trained using the last 30 days of covid cases from Covid19 API. We decided to only keep it to 30 so that it would adapt to downward trends if they were to occur. However we then faced the following difficitules:

1. When tuning the brain, we realised that each country is different and required different configurations.

2. When training the brain it required all statistics to be normalised to be between 0 and 1. This caused problems especially when normalising stats in the USA where it has over 1 million cases in some states, whereas some states in India only have a few hundred.

This meant when we denormalized the predictions (that were in the 0-1 range) back into the millions, small fluctuations would multiply into bigger fluctuations, which explains large steps in the graph when comparing recorded data and predicted data. These fluctuations are less obvious in the India graphs since they are only in the hundreds, and results in a smoother more realistic prediction. Due to this constraint we decided against the use of neural networks and decided to use our polynomial regression solution

3 Deliverable 2 (API design)

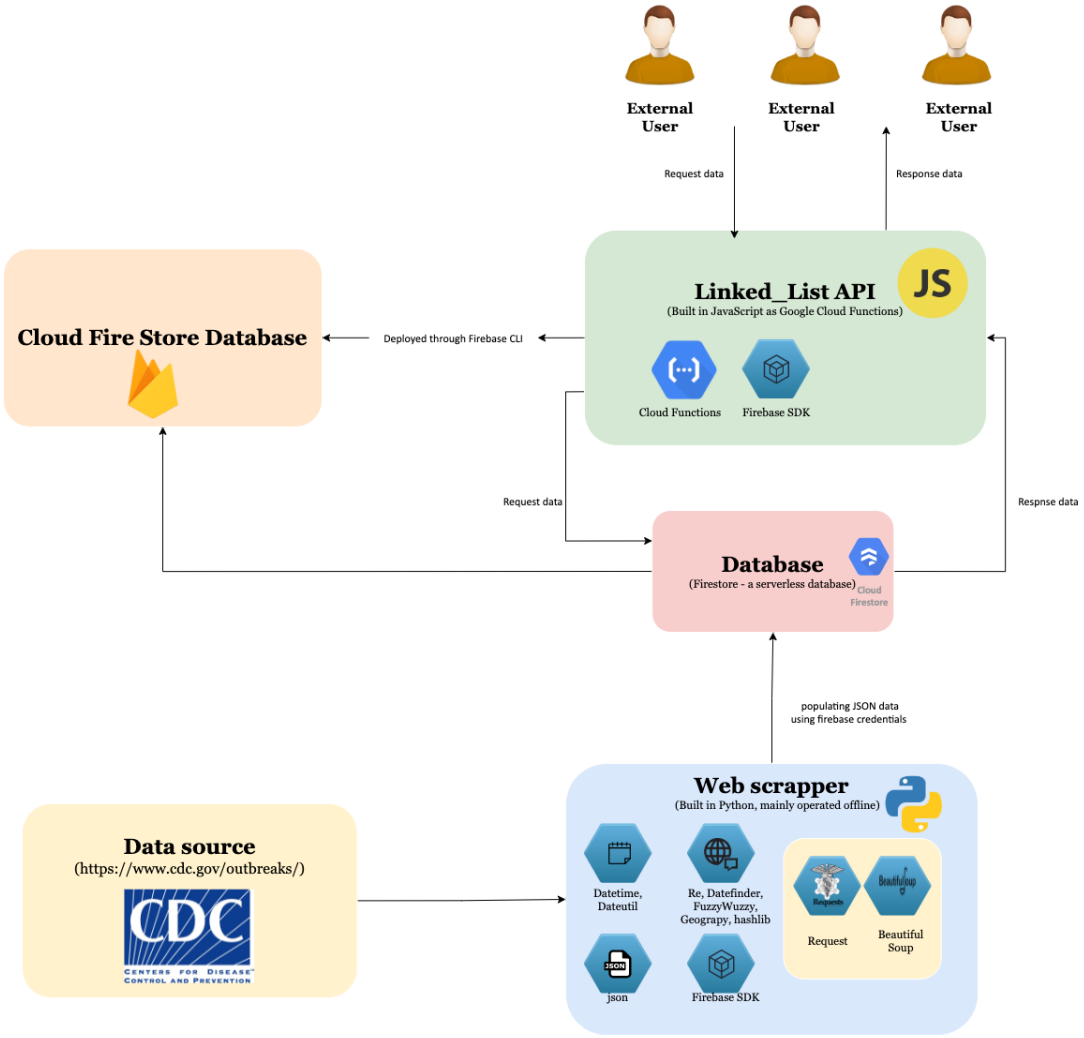
3.1 Final API Architecture

3.1.1 Overall API

Our API was built to be easy and quick to develop, whilst still providing a user focused product. This is a product that provides the user the means of getting the right data, when they want it, how they want it.

Diagram of our final API structure:

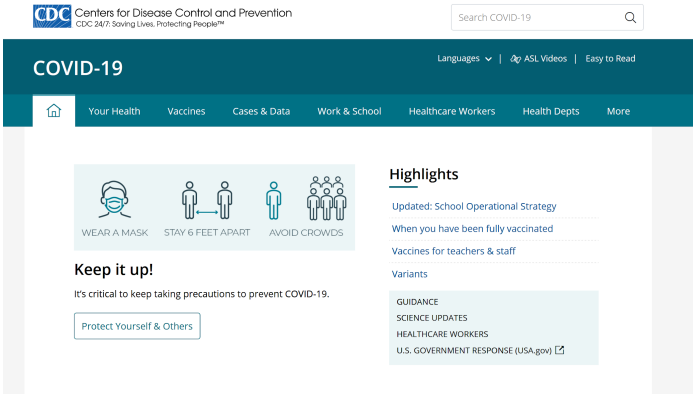
- For more explanation read below



3.1.2 Data Source - CDC Website

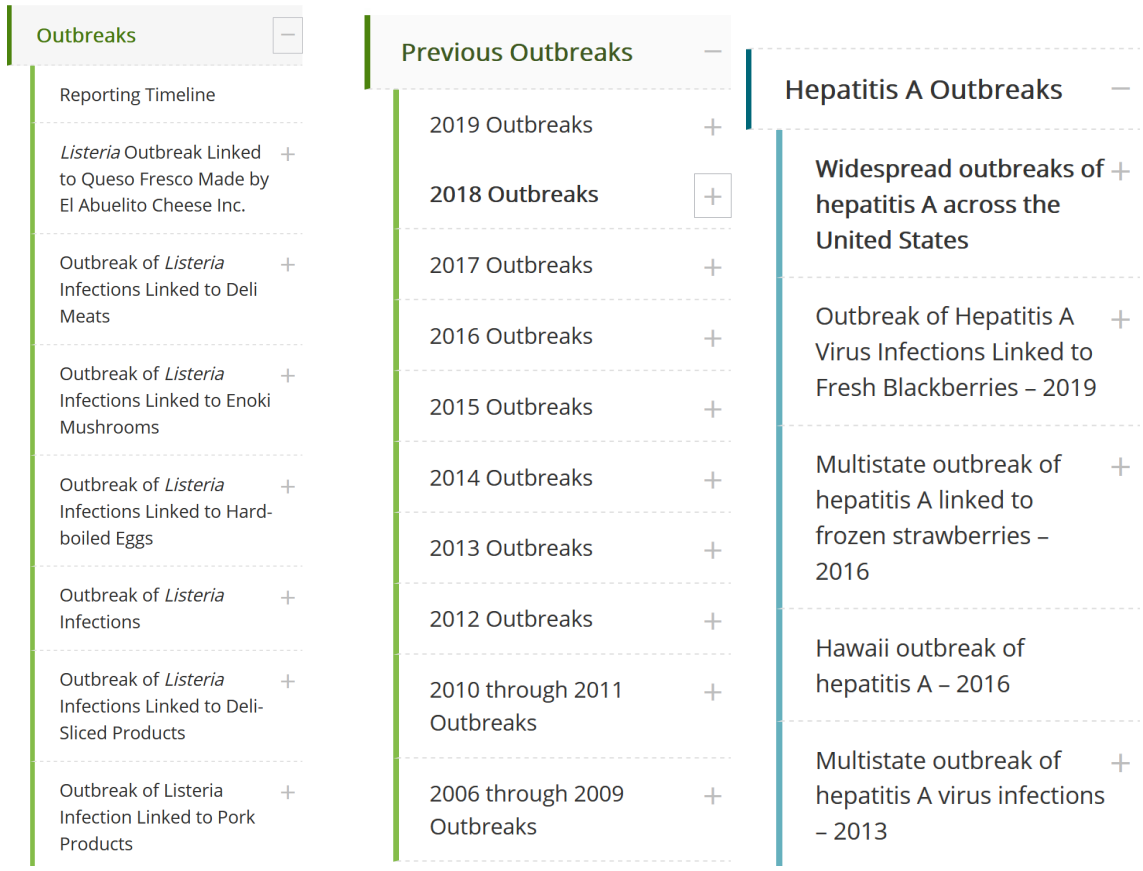
3.1.2.1 Challenges of this data source.

Our data source is the CDC. Since they do not have an API, our team uses a web scraper to retrieve the data. However this was a difficult data source to use because it has a few different page layouts (Shown below).



The above two images are examples of pages with vastly different layouts. As seen in the screenshot, the first page has a field for published_date shown on the page but the second one is somewhere hidden in the HTML. This means every time when we see pages with different layouts, we have to manually examine through their html and discuss the cases separately in order to extract the target information.

CDC webpages are not uniformly structured, and this situation happens more than we expected it to be. In general, information that has specific fields dedicated to includes ‘title’, ‘date_of_publication’, ‘url’ and ‘main_text’. They are relatively easy to extract. However, for information like “location”, “event_date”, “syndromes”, CDC does not have specific fields dedicated to them, therefore when scraping we will have to use other libraries or regex to gain results.



The above images is a comparison of the outbreaks section between Listeria, Salmonella and Hepatitis A. We can see that even though they have similar layouts, there are still minor differences in text. Using event_time as an example, if there are valid times shown in the titles, then we use them as the event_date. If not, then we have to examine through the main_text to find all of the phrases that look like a datetime, analyse their validity, then put one of them as the event_date.

3.1.3 Data Collection - Web Scraper

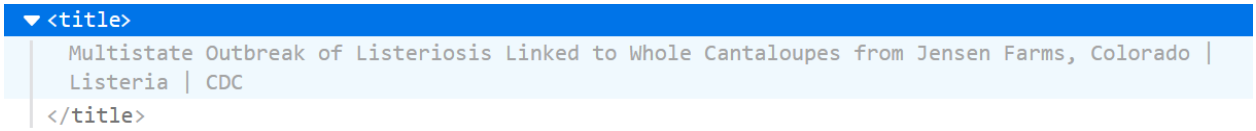
3.1.3.1 Implementation

The Web scraper is implemented in python using request and beautiful soup as the main scraping tools. Despite their poor speed performance, they are easy to use when it comes to identifying tags and extracting text. For scraping CDC web pages in particular, the functionalities they provide in extracting target information from static HTML pages are more than enough.

The procedure can be summarized as:

1. Collect all links, including the links on the base page as well as the nested links
2. Request for HTML page then using beautiful soup scrape information from it.
3. Collect the information we scrape and put them into corresponding fields or objects.
4. Upload the data onto firestore using Cloud Firestore credentials.

During the process of extracting data, we imported various libraries to help us achieve the goal. For most of the pages that we scraped, the fields title, main_text and publish date are very easy to retrieve using beautiful soup and regex. Title is usually in between <title> tags, main_text can be scraped by searching for <p> tags, and publish date usually has a field called DC.date. Here is a quick example of how title is displayed in the HTML:



For the information that is dynamatic over different pages, we collected them by utilising a variety of available python libraries - They include: Fuzzywuzzy (compare the similarity between two words), Geography and Nltk (extract location names from text), Datefinder(extract datetime phrases from text). Geography and Nltk are used to extract location information about outbreaks, datefinder is used to determine the earliest event of the outbreak, and Fuzzywuzzy is used to compare the words in a given text with the identifiers provided in the spec to determine diseases. Here is an example of how disease is determined using Fuzzywuzzy:

Title: Outbreak of Listeria Infections Linked to Deli Meats
Identifier list: [{name: cholera}, {name: listeriosis}, {name: COVID-19}...]

We split the title by whitespace then loop through each word and compare with the name in the identifier list, Fuzzywuzzy library will be able to tell that the word “Listeria” is similar to “listeriosis”, therefore have listeriosis as the disease of the report.

For diseases’ with more than 1 word, we likewise did the same for what we did for 1 worded diseases, made an identifier list to identify thus.

However, there are flaws towards using these libraries. They are detailed in challenges and shortcomings.

Deployment is analysed at the end of scraper section.

3.1.3.2 Challenges:

The most challenging part of building a web scraper is to be able to cope with different page formats while ensuring the accuracy of the extracted information. In order to achieve this, we manually compared pages with different layouts that our scraper might come across, and coded to ensure we covered as many cases as possible.
Another challenge was to come up with the best scraping strategy along with the imported libraries. For example, since CDC does not have a clear field for us to extract disease type, we needed to find a way to

scrape the result as accurately as possible. The solution provided is very effective, however it's effectiveness is limited to the size of our identifier list for each disease. Therefore to improve the solution increase the size of the identifier list or use a more effective natural language analysis solution (we have yet to find).

3.1.3.3 Scrapping outcome

Extracting location

To extract location we came across a Type I vs Type II error. We have tried two libraries Geotext and Geograpy. Geotext cannot do an in-depth matching. For example, it can't handle upper and lower cases and abbreviations. Due to these shortcomings of Geotext, we decided to use Geograpy. However Geograpy can be too in-depth sometimes. For example, it picks up the word "Ask" or "Standard" as cities. Thus, we had to manually exclude these cases, but similar cases may still occur, and lead to a result of having location with confusing names.

Event date matching

What we are doing now is first collecting datetimes from the title. If there is a result then we return the earliest time, if there is no result then we collect datetimes from the main_text. Again if there is a result then we return the earliest, if there is no result we then return the publish date. We are able to exclude invalid dates (e.g year being 800). However we can not ensure that the earliest date mentioned in the main text, is the starting date of the outbreak. We have made it so it is highly likely to be the case but it is not a perfect filter.

Disease matching

We compare strings of the title, word by word with the identifier list and return the word that has the highest similarity with the identifier list. It works most of the time but there are edge cases. We had to manually handle those cases and the issues mentioned above are mainly because webpages on the CDC are too dynamic and there are no specific fields for us to easily collect other than title, publish_date and main_text.

Long performance time

Since our scraper mainly uses beautiful soup and requests, along with the usage of many other libraries, it takes a long time to collect all the target data. It also affected our deployment - which is detailed in the deployment section.

3.1.3.4 How this differs from our initial design

In our initial design, we mentioned three points:

a. We were going to do a depth first search
b. We were going to scape everytime the API gets called
c. If no calls in a week it does it using cloud scheduler

For point a, during implementation, we realise that with the help of python beautiful soup library, we do not need to worry about the algorithm behind searching.

For point b, it is not reasonable since that discards the usage of firestore. It is also not realistically possible since the scraper takes a long time to run.

For point c, we are able but have not yet implemented due to the difficulties we are having with deployment. It is detailed in the Deployment section.

3.1.4 Deployment

3.1.4.1 Implementation

For web scraper, we initially had 3 ideas.

1 idea

- When user calls the API
- We call the web scraper endpoint
- Goes to the database to find the relevant URL's that need to be scraped
- Compare the last scraped time with the last modified date of the HTTP request
- If page has been updated, then webscraper will scrape the page
- Updating the database

2 idea

- Google cloud scheduler
- Calls the function every 2 hours to make sure that we are up to date all the time

3 idea

- Every 2 hours via google cloud scheduler, the API is called we can run a checking endpoint to see if the website has been updated, if no then we do nothing
- If not we can go scrape it

During deployment, we decided to go with the second idea. This is because realistically we are not able to call the webscraper every time when users call API due to the long response time of web scraper, and it discards the idea of having a database. We are also short of time to implement the "last scraped" functionality. Additionally, CDC does not update their website very frequently, therefore 2h is more than enough to ensure the data in our database is updated.

3.1.4.2 Challenges:

The main challenge for now is that we cannot deploy it on google cloud due to the time limit they have. Google cloud functions only allow a 9 minutes runtime, however our web scraper takes around 20

minutes to finish. This is the biggest challenge for now, and we are still working on it. Some solutions include breaking scraper into different parts, using App etc.

3.1.4.3 Shortcomings

We only partially deployed the web scraper - it only worked with collecting “article” object but not “reports” or “locations” due to the runtime limit google cloud has. We are currently working on breaking web scraper into parts and deploying them separately.

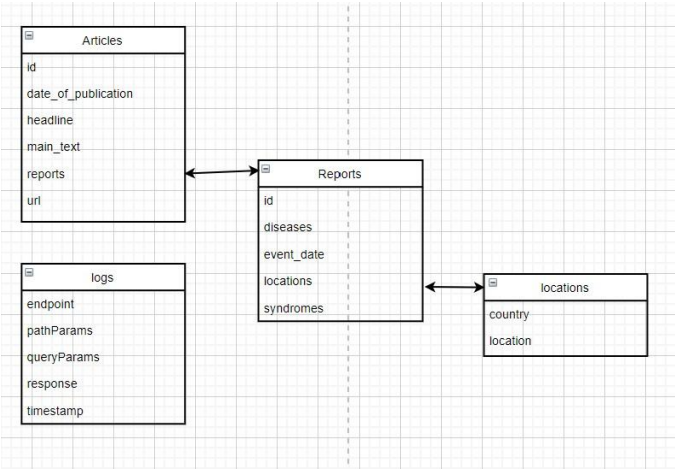
Furthermore other short comings include:

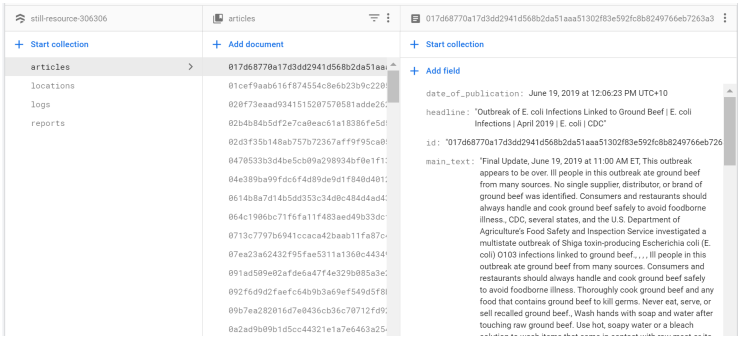
- Underestimating the time and knowledge it takes to deploy to any cloud platform which has resulted in a half deployed web scraper
- Not taking into the account the many requests we do towards the web page, which takes up the majority of the runtime and hence why our scraper takes on avg 20 mins to complete (complete being scrape the CDC and then upload to firestore)

3.1.5 Data storage - Firestore

3.1.5.1 Implementation

The diagram on the left uses entity diagrams to describe how we are storing our data in Firestore.





We implemented our data storage through Google’s Firestore. After we gather the data from https://www.cdc.gov/, we store it in this NoSQL Database. We stored our data in this type of database because it allows related data to be nested within a single data structure. This allows us when we are developing, to access the data very easily, easier than having a traditional fully relational database which requires complex table operations. In addition to this, firestore has a very easy to use user interface. Allowing us to easily understand how our data is stored and how to extract it.

Within this database we have four different collections of data: articles, locations, logs and reports. These are based on the endpoints we are going to implement. A general API search with period of interest, key search terms and location, we will just search through the articles collection to find the data. However if you are looking for a specific report or location we just quickly search our reports or location collection. Finally, the logs store the API search history. This means the API user can see what other people are searching. This method of storing data per endpoint, allows us to make the most of the NoSQL databases benefits, allowing us to easily return the right data to the user.

3.1.5.2 Challenges

None we found cloud firestore database very easy to use.

3.1.5.3 Shortcomings

At the beginning we did not think about how regions or zones would affect our firestore. This lack of understanding made us host our Firestore database in the us-central zone. This means that the data is far away from our end users which are based in Australia. We are currently migrating over to Australia-southeast, however it is not yet complete. Since the majority of our users are based in the australia-southeast zone then hosting our web service here leads to lower latency and higher bandwidth to deliver content to our users faster.

3.1.6 Data service - Google Cloud Functions

To create a web service, we had to first think about what makes a web service good. Availability, ease of use, standardised input and output, using appropriate nouns to describe endpoints, and nested resources e.g. articles?id=1 and gracefully returned standard error codes are needed.

3.1.6.1 Overall Implementation

We chose to use Google Cloud Functions to host our web service. This did exactly what we expected, and allowed the data to be available all the time. In addition to this, since Google Cloud Functions, is inextricably linked to Firestore, this made development very easy. We were able to use the Cloud Firestore CLI to deploy our cloud functions, and the functions logger makes development very easy to debug.

We implemented the API web service using NodeJS. We had to switch from python3 because NodeJS was the only supported language in Cloud Firestore Functions. This mistake arose because we were under the impression that Google Cloud functions supported Python3 and Flask.

Advanced Querying

This feature set is applied to the /api/articles endpoint. The following allows the user to query the API in specific ways in order to get a more refined data set, specific to their needs.

Period of Interest

We had an interesting challenge with how we allow the user to input date strings. There are two different ways to input a date - a date range string or a singular wildcarded date string.

- DATE RANGE: when providing a date range, neither date (start or end) may contain wildcard characters (i.e. ‘x’). Returned articles are within the start and end dates inclusive.
    - e.g. “2019-01-02 00:00:00 to 2020-01-02 13:00:00”

- WILDCARDED DATE: when providing a wildcarded date, the date must contain at least one wildcard character (i.e. ‘x’), which represents any number from 0-9. Since wildcards can represent a range of values, a wildcarded date is in itself a query.
    - e.g. “2019-01-xx 12:xx:xx”

We allow the user to input data in these different ways, to give them more flexibility for their different use cases. For example, the wildcarded date allows a user to specify a year very quickly. They would only have to input:

“2019-xx-xx xx:xx:xx”
Rather than the lengthy:

“2019-01-01 00:00:00 to 2019-12-31 23:59:59”.

Allowing our end user to develop and call our endpoints in a more effective manner, whilst still allowing them to have the precision of the date range.

Key Terms
Users can also filter by providing search terms, known as ‘key_terms’. The user provides a list of comma separated strings to the API and will return all matching articles that contain ALL key terms. Key terms are also not case sensitive and trailing white spaces are removed before being used so that insignificant modifications of the word do not inhibit the user’s usage of our API.

An above and beyond feature we implemented is the ability to retrieve articles that exclude key terms as well. This feature is easily used by prefixing a ‘!’ before the search term. Furthermore, these excluding key terms can be used in conjunction with normal search terms to provide the user the ability to retrieve finer search results.

For example, if key_terms=”corona,!salmonella,flu,!sneezing” then this will search for all articles that satisfy all four criteria:

- CONTAINS “corona”
- DOES NOT CONTAIN “salmonella”
- CONTAINS “flu”
- DOES NOT CONTAIN “sneezing”

Location
Lastly, users can refine their search by specifying a location of the reports in the resulting articles. By providing a string, this will return any articles that contain reports containing the provided location. This is also not case sensitive and removes trailing white spaces to ensure a more efficient user experience.

3.1.6.2 Shortcomings

There are several shortcomings to do with our API currently. Right now, we do not allow for pagination from requests, we only slightly filter before getting the resource from firestore, various security issues and do not have caching no versioning implemented. However, our web service meets our current sprints goals: to provide the user a functioning API that they can access at all times, and query using various endpoints and parameters.

Pagination
In terms of efficiency, we do not have pagination, which entails that the user could request a smaller amount of search results leading to a faster response time.

Filtering before querying firestore
After receiving the API request, we could use this request exactly to query the Firestore database more effectively. Currently we have a filter that only retrieves entries from the date range specified. For example if the path parameter period_of_interest was:

“2019-xx-xx xx:xx:xx”

The request to Firestore would be any articles within this date range:

> “2019-01-01 00:00:00 to 2019-12-31 23:59:59”.

We could filter the resource more before calling the firestore base. We could filter it by location and key terms. This means the processing will be faster because NodeJS does not have to receive the barely filtered database every query.

Caching
Next is caching, which would bring the data closer to the user again, if someone else nearby or that user has requested that data before, it will not have to get the data from the server again, but rather, the data is located in a much closer location. Again, increasing the response speed. This is a simple fix about adding a few headers, so will be completed in the next sprint.

Security
In terms of security, we have chosen a very loose cors policy, currently accepting all other websites. This is because we have not implemented OAuth or other authorisation, and our priority is to make the data

as accessible as possible. In addition to this, we do not have any POST, PUT or DELETE requests, so an attacker should not be able to alter the database, however we could be attacked with a denial of service, forcing us to go above the free tier of Cloud Functions and have to pay for these requests. A way of stopping this is to have users login and rate limit them. This means if a user (that is logged in) queries our endpoint 100 times in a minute, it is probably an attack and not a user actually trying to get the data. Instead of letting them complete these requests, we can not allow this user to send any requests for a time limit that grows exponentially. For example, the first limit can be 1 minute, then 2 then 4 and so on.

Versioning
Finally, versioning is key to allowing the user to adapt to change when they are ready and allow us to test versions of code before releasing it to the user. However, this does not matter as of yet, because we have zero users, but very soon we will implement this.

Errors and outages
In terms of error handling and outages, Google cloud also allows us to host our service in multiple regions so that in the case that the servers in one place go down we can handle the outage

Future Implementations
In the future, the above shortcomings will become our backlog of tasks to do.

3.1.6.3 Challenges

The API was harder to implement because we are not as familiar with NodeJS and Javascript as we are with python3.

This is due to the easy to learn nature with the google cloud platform

3.1.7 API Response

3.1.7.1 Implementation

We are building a web service for a group of people that are united by similar API responses. Therefore we should also build our API as close as we can to this design. This would make it easier for our users to integrate our product into their different use cases. This is due to the fact that they are already familiar with the response.

For example the following search returns:

GET:
https://australia-southeast1-linked-list2.cloudfunctions.net/api/articles?period_of_interest=2019-01-11 01:32:xx

```
{
  "log": {
    "team": "SENG3011_LINKED_LIST",
    "time_accessed": "2021-03-26T04:15:15.892Z"
  },
  "articles": [
    {
      "id": "558f2efbbd7501579437a3fe8bab5e3f958d7e4f113ed389681f8f4cae21e4cf",
      "date_of_publication": "2019-01-11T01:32:15.000Z",
      "url": "https://www.cdc.gov/ecoli/2008/ground-beef-kroger-7-18-2008.html",
      "main_text": "NOTICE: This outbreak is over. The information on this page has been archived for historical purposes only an
      "headline": "July 18, 2008: Investigation of Multistate Outbreak of E. coli O157:H7 Infections | E. coli CDC",
      "reports": [
        {
          "locations": [
            {
              "location": "New York",
              "country": "United States",
              "id": "9b5a85e97e956a1e337873c89d65f495cca8a70db6ca8f9c8da34737f64e3954"
            },
            {
              "id": "7104b63491def7a43494dae9ce7db2c0db0c0f63b8cac1ca06a6d5edefe85ee4",
              "location": "Indiana",
              "country": "United States"
            },
            {
              "id": "932005b7d7b235a1cb13a5edd139ce17c20f73ea4fd2ce24b62e523c59e29532",
              "location": "Ohio",
              "country": "United States"
            },
            {
              "id": "a0973d48f951d159958a59dcdff8fe817df6ff1c2443f224d399b076f68fac6a",
              "country": "United States",
              "location": "Michigan"
            },
            {
              "country": "Georgia",
              "id": "29758482b1b7b53689d5ff399979db011e2fc2260f89fb0f69e5e3e7b31d0f9a",
              "location": "unknown"
            },
            {
              "id": "378beee5676359235d37951c5e9d34ee1fd0fa7816088e7b602b033d45ed4947",
              "location": "unknown",
              "country": "United Kingdom"
            },
            {
              "location": "unknown",
              "country": "Brazil",
              "id": "60fbe59ef71c77cabba3e1a72c61334b529f8aa8f260c371eeb76f08be464b9d"
            },
            {
              "location": "unknown",
              "id": "702ad8c1123005534be05425cb45231d5f3be9b751e255fbb13c0b13ec9545e1",
              "country": "Jamaica"
            }
          ],
          "event_date": "2008-07-18T00:00:00.000Z",
          "syndromes": [
            "unknown"
          ],
          "diseases": [
            "ehec (e.coli)"
          ]
        }
      ]
    }
  ]
}
```

3.2 Test Documentation

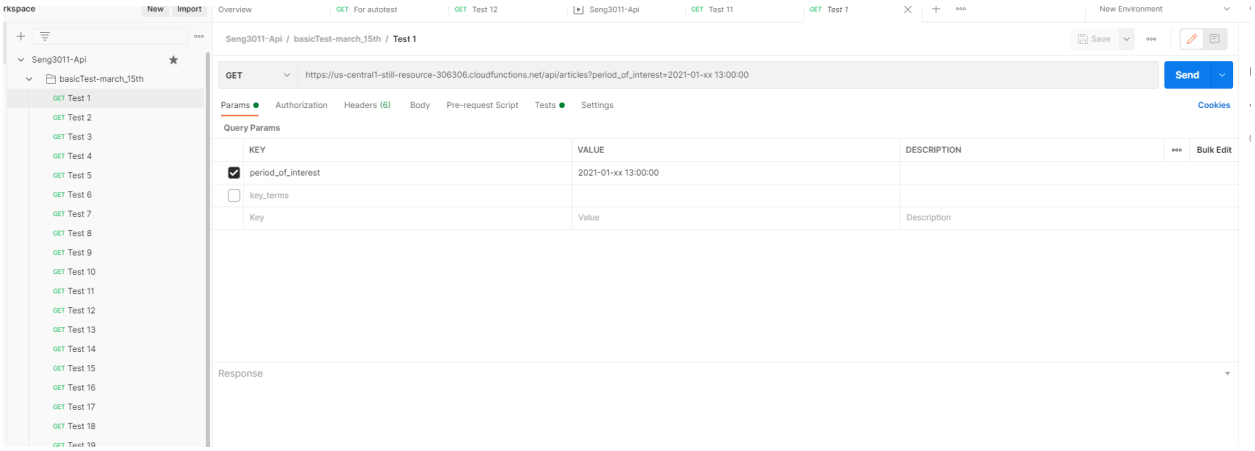
3.2.1 Unit tests for API

3.2.1.1 Test Brief

The purpose of our testing is to ensure our API is working as intended, which is to provide data from a database which is populated using a web scraper on the Centers for disease control and prevention website.

Our unit tests are mainly implemented through postman. Postman is a test software that can test automatically. It allows us to define URL, request mode and query params freely. At the same time, it provides script functionality for automatic tests, as well as multi clock view return data. We can view the return value of our API from page and file, so as to simply query the expected output of JSON. With it, we can simply create one or more requests and specify the corresponding test set in the corresponding test. At the same time, it provides run-time detection, so that we can detect the efficiency of our program.

It also supports the command line run mode based on Postman and Newman, which makes it easier for us to run automated tests to test our programs.



3.2.1.2 Test Input

Test ID	Test reason	Test analysis
1	Basic test-corresponding to a single timestamp	Success

2	Corresponding to two timestamps	Fail
3	Corresponds to 1 precise time	Fail
4	Corresponding to 2 precise time	Success
5	Corresponding to 2 precise times (standard)	Success
6	Corresponding to 2 precise times (reverse order)	Success but no information, an error should be reported
7	Corresponds to 2 precise times, but the time is wrong	Return 200,but need 400
8	Corresponds to 2 precise times, the first time is wrong	Success
9	Corresponds to 2 precise times, the second time is wrong	Success
10	Corresponds to 2 precise times, ignoring hours	Success
11	Check Extremely close time	Success
12	Check Same time	Success
13	Add the normal Test -cov of key_term	Success
14	Abnormal Test-cov with key_term added	Success
15	Add the normal Test wuhan of key_term	Success

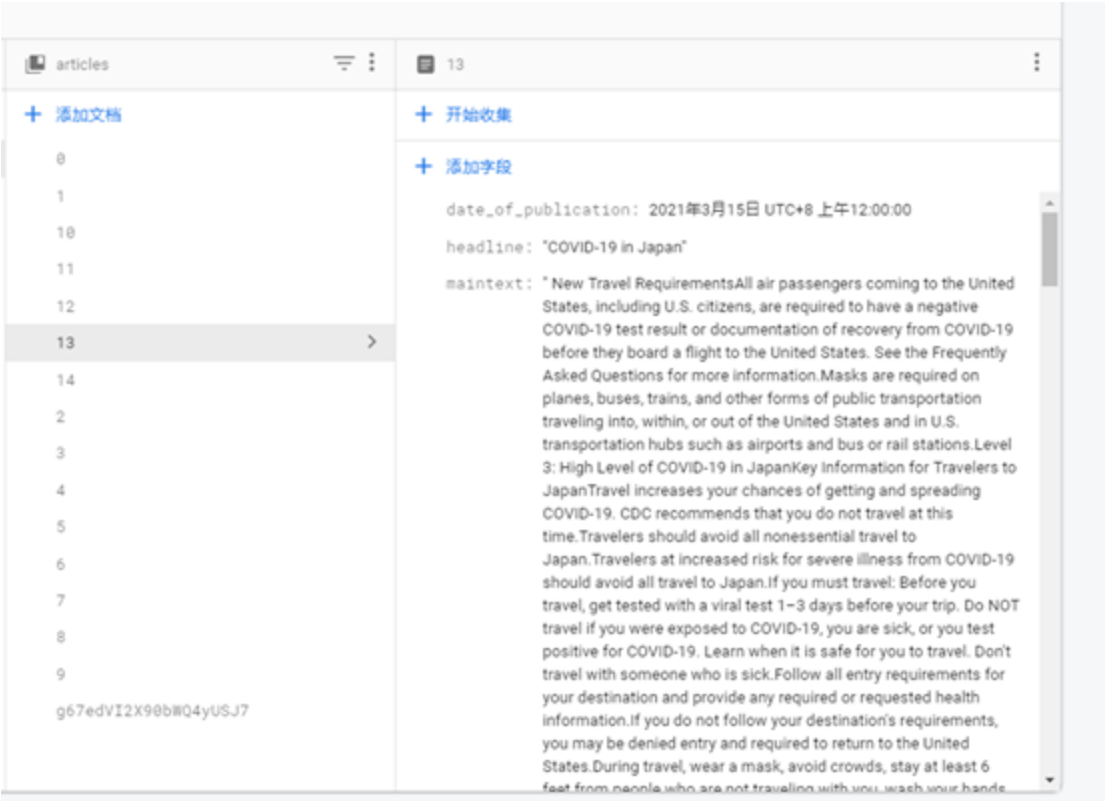
16	Add key_term's normal Test Fever of unknown Origin	Success
17	Normal Test fever with key_term added	Success
18	Add key_term abnormality Test silence wench	Maybe it should return an error response that no information was found, but 200 was returned, need 400
19	Can the test recognize similar diseases -covid-19	It seems that there is no such function

The data of Unit test in Phase_1/TestScripts

3.2.2 Test for Scraper (Manual tests)

We test whether our scraper runs successfully by giving the specified web page. First, we find a web page whose information has been manually determined (the amount of information that can be captured is less than 20, so we can simply check it), and then we execute our scraper. When we run our program, it will automatically upload articles and corresponding reports to the cloud server, namely firestore, and then we will screen the data by calling the functions inside and finally determine whether our data is correct.

Test ID	**Testing Reason**	**Test Analysis**
1	Standard CDC webpage	Success
2	Standard CDC webpage with accurate event time	Success, Returned the correct time
3	Standard CDCD webpage with accurate event time in the future	Success, the default value is returned



3.2.3 Performance and Load Testing

We use Stoplight and JMeter for load testing, as these were the most convenient to use in our set up with cloud endpoints and being that stoplight was where we wrote our documentation. Using these tools we analyze the load of our program by simulating two scenarios: multiple different requests and multiple user requests at the same time.

L1 When testing the impact on our API based on multiple different requests,

We chose stoplight as our testing software, as we wrote our documentation on this platform. Through the use of it's own preprocessing function, we wrote a program to automatically generate random query time and key terms script for terms. The lexicon of key terms comes from the disease collection in spec. The test calls randomly generated queries many times.

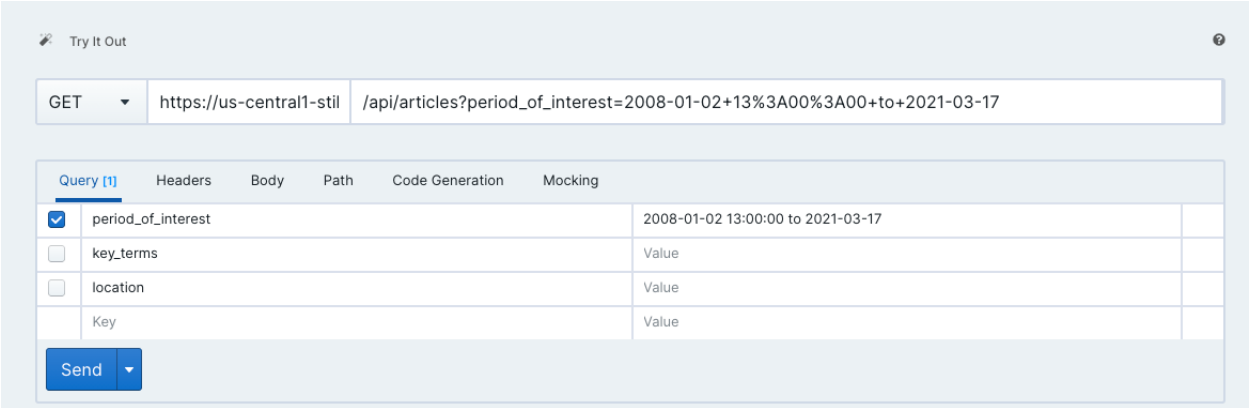
```
var random = GetRandomNum(0, 86);
var query = locationwords[random];


var period_of_interest = firstTime + " to "+ SencondTime

pm.environment.set('period_of_interest',period_of_interest)

function GetRandomNum(Min, Max) {
    var Range = Max - Min;
    var Rand = Math.random();
    return(Min + Math.round(Rand * Range));
}

var locationwords = ["China","Japan","Us","UK","Asia","Australia","German","India","Haemorrhagic Fever" ,
"Acute Flacid Paralysis" ,
"Acute gastroenteritis" ,
"Acute respiratory syndrome" ,
```

Test ID	Test input	Number of request	Avg time (ms)	Max time (ms)
1	Return time period	10	302	560
2	Random time period	50	310	736
3	Random time period	150	320	1200
4	Random time period with random key	10	393	461
5	Random time period with random key	50	310	600
6	Random time period with random key	150	311	810
7	Random time period with random key	200	340 (has error)	1297

As far as testing is concerned, large-scale queries in turn in a short period of time will not affect the running speed of the API too much. Queries less than 150 times are about the normal load capacity of the API. When the number of requests exceeds 200, some errors will be generated, this is due to the API not being able to handle such large requests as it times out, this is due to the Google Cloud Timeout limit we set on the API calls

P1 Test how long it takes to query the entire database

Input	Time
period_of_interest=2008-01-02 13:00:00 to 2021-03-17	7.35s

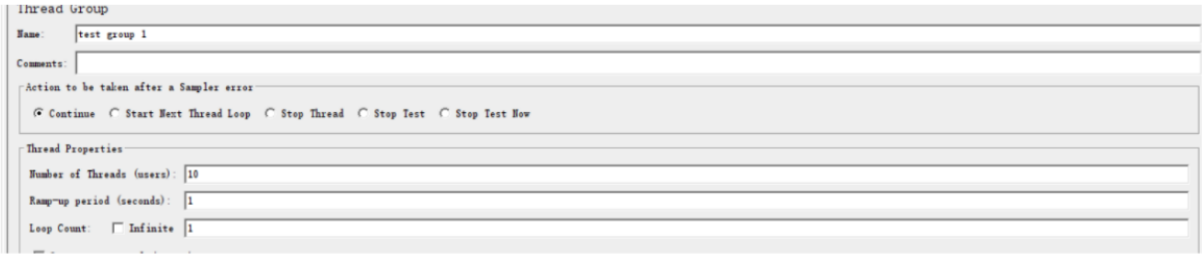


P2 Test how long does it take to query one thing

Input	Time
period_of_interest=2020-01-16 13:00:00 to 2021-01-16	300ms

L2 Performance under multi person processing.

Since postman only supports single threaded queries, I use JMeter as a tool for multi-threaded queries.In this test, I tested the performance of the API when multiple users access the API at the same time in a short time.However, since the test is conducted overseas, the delay may be large.The actual situation will be much better (almost one in five).



Input	UserNumber	Average time (Including transfer and load)	Error rate
period_of_interest=2019-01-10%2013:00:00%20to%202021-05-20%2005:44:23	10	2393ms	0%
	50	3421ms	0%
	150	4498ms	0%
	180	5600ms	2.2%
	200	7579ms(due to error)	3%
	500	8710ms(due to error)	65%

According to the data, the maximum load of API is about 150 users, and the return speed will decrease with the increase of users.

3.2.4 Security Testing

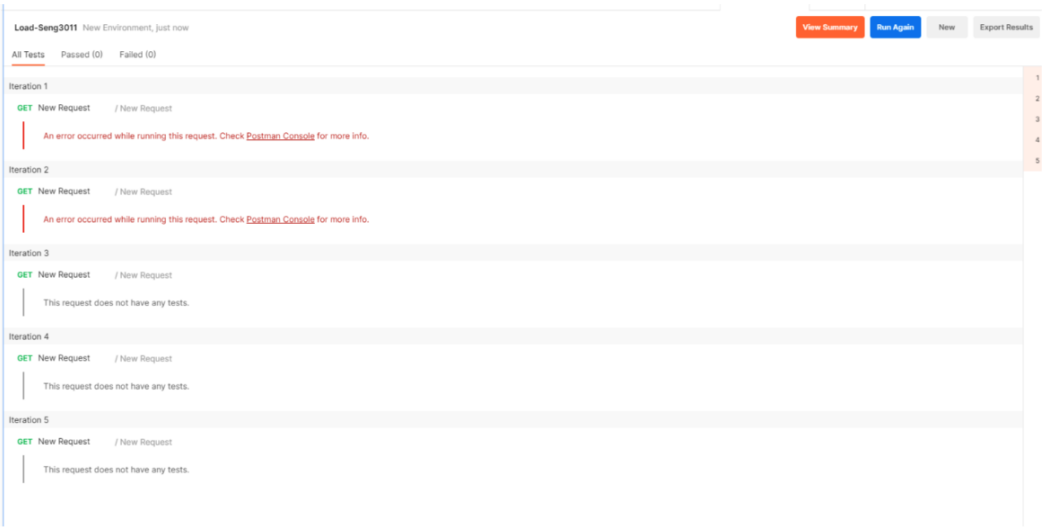
We test API security, on the one hand, to make our users get the right information while ensuring our API provides reliable results on a consistent basis .

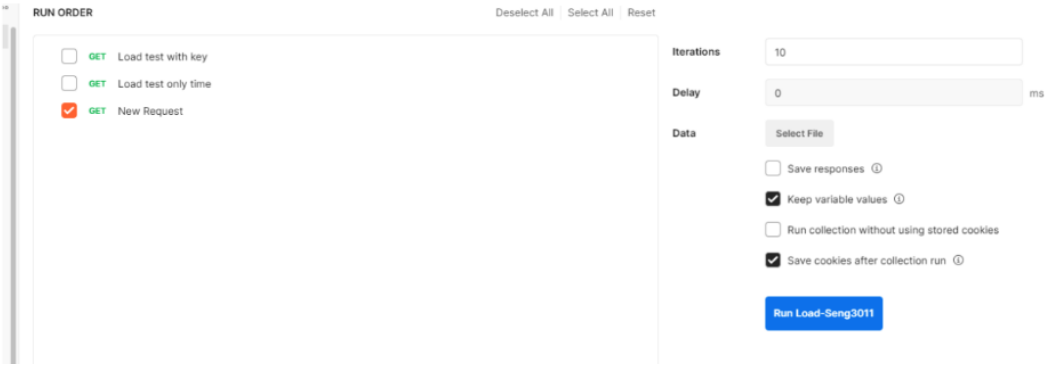
Because our API only provides the query service to the database, and does not save any user information in the API, we only test reliability of the serverless functions

Google Platform has a simple feature called adaptive protection, which relies on machine learning models to detect incoming DDoS attacks. When I use postman or JMeter to make 50 requests to the remote server in 0.20ms, the server rejects my request, thus showcasing the Adaptive Protection feature.

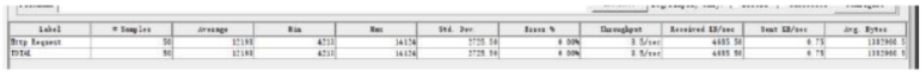
Thus as a result it showcases the reliability of our API and it’s resistance to DDoS attacks, however attacks effective against Google, may and can be effective against us. Therefore we should look for other features for security

In addition, the data based on the load test shows that when the API receives a large number of visits from the same IP in a short period of time (about 180), the system will automatically reject some requests from this IP.

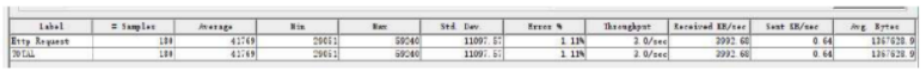




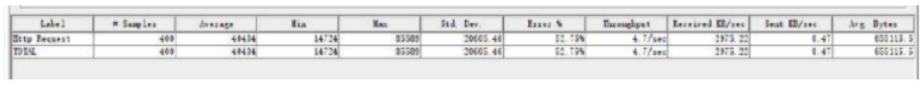
150 request



180 request



400 request



3.2.5 Limitations

Firstly, our automatic tests do not run synchronously with API iteration, and it only provides the test of current functions. If we add new functions, we need to rewrite the automatic tests and redeploy them.

Detecting the performance of the scraper is a very difficult thing, because it is difficult for us to make sure that the data scraped is accurate, as it is based on CDC's website structure. We can only ensure the accuracy of the information by manually checking the database. At the same time, if we want to ensure that it is accurate, we need a better scraper to provide better accuracy.

In addition, for security testing, we only tested the API response measures under large-scale traffic attacks. We have no way to detect the attack method of implanting Trojan horses to obtain user information, because our API is only an API for obtaining epidemic information. It does not contain any user information and/or private information.

The performance test may be biased, because each person may have different physical distances and transmission speeds based on different people's network speeds or different regions, resulting in different delays. We can only try to estimate the internal running time. Thus giving an expected value, we did this to get a wide range of results, while also not having the man power to allocate more time to testing due to our situations and circumstances.

4 Deliverable 1 (Pre-design)

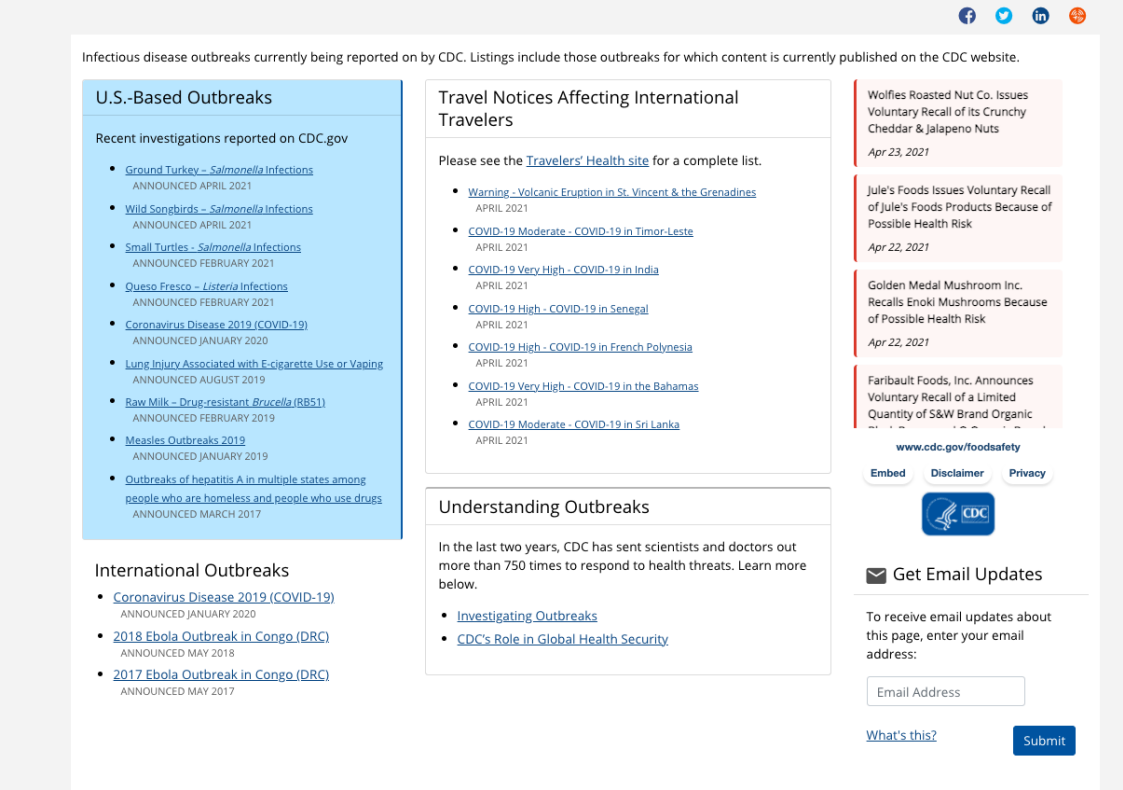
4.1 Describe how you intend to develop the API module and provide the ability to run it in Web service mode

The development of the API Module will be divided into 4 steps:

1. Data Collation
2. Api Architecture Planning and Data Restructure
3. Making the data available
4. Testing API

Deploying this API module to Google cloud functions with access to firestore, will allow it to run in web service mode.

4.1.1 Data Collation



We will be using the BeautifulSoup4 Python library to build a Python web scraper to extract data out of www.cdc.com.gov. The scraper will automatically follow relevant links on https://www.cdc.gov/outbreaks/ using a depth first search approach and collate all information

regarding a particular disease together into one object. A depth first search would allow us to collate all relevant information with regards to a specific disease report. The reason for choosing beautiful soup over other libraries is illustrated in Q3.

Due to our inexperience in cloud platforms and not understanding how deployment works we went with a simpler solution, which was using google cloud platform's cloud scheduler to call the scraper weekly. A weekly schedule ensures that our database contains the most relevant information. This could have been better implemented with the use of a cloud function that would check the headers of each page to determine if it was modified recently, instead of wasting resources weekly to update our database.

Furthermore due to the infrequent nature of the articles being updated in an irregular fashion we decided to go with a weekly update, as opposed to a monthly, due to the lag of getting resources and as opposed to daily due to the over use of cloud resources.

4.1.2 API Architecture Planning and Data Restructure

Once all information has been scraped, we will inspect the data objects together as a team and design the query response architecture that accommodates for all disease reports (e.g each disease report will have at least a generated id, name, location, etc. as per the specification).

We further want to enhance the user experience with additional endpoints other than the search endpoint, the additional end points include:

Endpoint	**Description**
/api/article	Gets the article that corresponds to the input id
/api/articles	Endpoint for getting reports from, period of interest, key disease terms and location.
/api/logs	Gets the last 10 20 API calls

For more information refer here: https://seng3011.stoplight.io/

Once the data has been scraped via cloud platform it will be converted to JSON then populated into Cloud Firestore's Cloud Firestore using POST method. Cloud Firestore's Cloud Firestore is a cloud database used to store data online. The screenshot below is an example of how we are planning on populating data into Cloud Firestore with cloud functions.

Since we have chosen to go with Cloud Firestore's Cloud Firestore free trial plan (see a comparison between Firestore and realtime database in part3), we are also faced with the storage limit of 1GB on the free hosting storage plan. If the information we have scraped from www.cdc.com.gov exceeds this limit, we will restructure the data collected from the web scraper by removing unnecessary information that we chose not to include in the query response, or duplicate information that would take up unnecessary storage in our database and bloat the response. This restructure would also require changes to the web scraper so that scheduled scrapes follow the new data format.

4.1.3 Making the data available

Once the data is available in the Cloud Firestore, we will host our API on Google Cloud Endpoints so that the service is web based. Making this service web based will increase its accessibility since the API will be queryable at any time. Furthermore, by making use of the Google ecosystem in Google Cloud Endpoints and Cloud Cloud Firestore results in a simple process to pass data around.

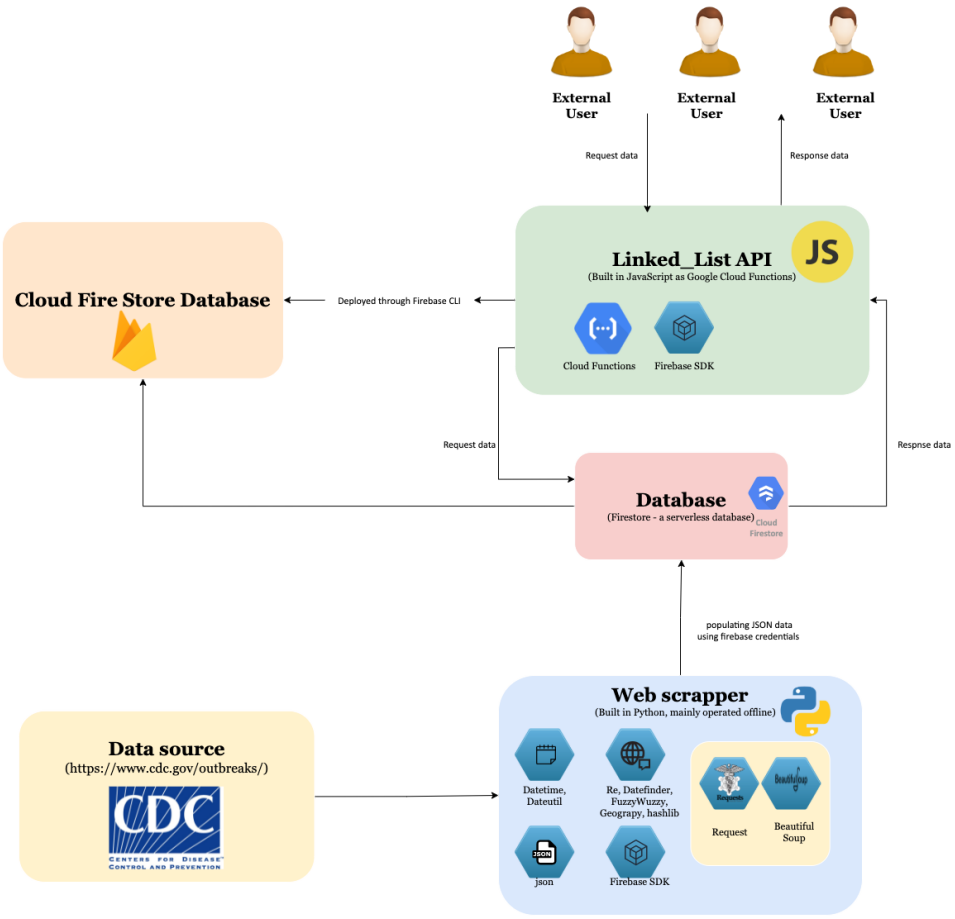
We will be updating the database using PUT cloud functions to make changes to objects that are already in the database. We will be updating it either on a regular basis (every 10 mins) or checking the last modified date to keep it up-to-date. We will also be building multiple GET methods with different parameter input combinations for clients to exact information from firestore. We are considering adding an API key for authentication. Users' allocated API keys will be examined in case they exceed the limit of calling our API or do not have a valid authentication. This can also prevent our API from malicious cyberattack.

The API will be written in Python since all members of the team are experienced with the language. (See below a comparison between Python and Java as to why we chose Python in question 3).

4.1.4 Testing API

Pytest will be used for testing. We will also use Postman to test our API in terms of checking responses with provided parameters.

Below is a general outline of our design, including how web scraper interacts with Firestore using cloud functions.


External User
External User
External User
Request data
Response data
Linked_List API
(Built in JavaScript as Google Cloud Functions)
JS
Cloud Functions
Firebase SDK
Cloud Fire Store Database
Deployed through Firebase CLI
Request data
Respnse data
Database
(Firestore - a serverless database)
Cloud Firestore
populating JSON data
using firebase credentials
Web scrapper
(Built in Python, mainly operated offline)
Datetime, Dateutil
Re, Datefinder, FuzzyWuzzy, Geograpy, hashlib
Request
Beautiful Soup
json
Firebase SDK
Data source
(https://www.cdc.gov/outbreaks/)
CDC
Centers for Disease Control and Prevention

4.2 Discuss your current thinking about how parameters can be passed to your module and how results are collected. Show an example of a possible interaction. (e.g.- sample HTTP calls with URL and parameters)

4.2.1 Overview

When using an API we must be able to pass arguments in and get return outputs. However, we must also consider the possibility of invalid input or internal errors, and thus we must also have error handling. Results will be collected in JSON as shown in the developer example, and presented nicely formatted when we develop the frontend.

4.2.2 Headers

There are a few different headers that we could add to our project to make it more secure, more efficient and make sure that it is up to date. The following is some of the request headers that we will use.

Header	**Value**	**Justification**
:method:	GET	Required
:scheme:	https (for security)	Required
:authority:	[[our domain name]]	Required
:path:	[[path to the requested resource]	Required
accept:	application/json	Our backend and other teams backends will be all built with this because it was specified and it makes our API compatible with other teams frontends
Cache-Control:	max-age=0, must-revalidate	Caching would allow the frontend to retain data without having to send a request all the way to the backend. This header specifically, means that the browser will have to validate the cache every GET request, ensuring our user has the most up to date data

Sec-Fetch-Mode:	cors	Our requests will be cors because we will be using google cloud functions, and that is not from the same origin as our website.
Sec-Fetch-Site:	cross-site	Our requests will be cross-site because we will be using google cloud functions, and that is not from the same origin as our website.

Now onto the response headers.

Header	**Value**	**Justification**
content-type	application/json	For best practice, the API will return the data as a JSON format, enabling front end teams and any other users to easily use our API
last-modified	[[date that was stored with the report]]	This means that the frontend or user that is accessing our data knows whether or not it needs to get the full load of data. If the data in the cache was last modified on the same date as the backend data then there is no need to send the object. This increases efficiency of the frontend whilst making sure that the user will always have the most recent data.
date	[[date object]]	Required
content-length	[[calculated from object]]	Frontend or user knows how much data it is about to receive. This should be good enough, rather than the connection header or byte ranges because the data that we are sending is not very large.
x-xss-protection	1	Stop cross site scripting attacks, and increase security for the user.
Content-Security-Policy:	default-src 'self' https://region-proj	When we add authentication and logging in features, this will stop a bad actor from sending requests to other sites that are not predefined by us. These security

	ect.cloudfunctions.net/function	headers mean that it will be harder for hackers getting a hold of any usernames or passwords, as well as performing cross site scripting on our users.

Overall, the use of specific headers will prevent some malicious behaviour, increasing efficiency and relevance of our frontend and making sure our end user has a better overall experience.

4.2.3 Parameters

We need to use specified parameters, so that our backend will know what specific data we want to retrieve from it. The API module will take in the following variables

Information JSON value	**Key Data Type**	**Justification**
location	ISO 8601 String	- Simple and easy to access
period_of_interest	ISO 8601 String	- Has its own library of functions to compare and what not - 3rd party library, enabling us to reduce possible areas for bugs
key_terms	ISO 8601 String	- Multiple terms can be easily stored in thus and can be found by incrementing through the array fairly easily.

We do not have an API key.

Considering that the API can be accessed via our website and/or through the web we decided to store the parameters in a JSON file as it is convenient and simple to pass around, as opposed to simple plain text, which can be seen by the comparison below between our rationale behind selecting JSON (*which is still plain text, but formatted in such a way that libraries have been built to make the use of thus more convenient)*

Property	**JSON**	**XML**	**YAML**	**PLAIN text**
Easy to read	✅	❌	✅	✅
Teams familiarity with data format	✅	❌	❌	✅
Simplicity (as opposed to complexity)	✅	❌	✅	✅

Stores Data	✅	✅	✅	✅
Ease of use in Javascript	✅	❌	❌	❌
Ease of use in functions	✅	✅	✅	❌
Ease and reliability of use with libraries	✅	✅	❌	❌
Ease of use in python	✅	✅	✅	❌

The table shows data properties that we as a team find important, as such we have decided upon using JSON

4.2.4 Example of Using the API

You do not need an API key to access our API.

As a user you can query the API via the end point here:
https://australia-southeast1-linked-list2.cloudfunctions.net/api

However for general use and simplicity use our stoplight API documentation linked here:
https://seng3011.stoplight.io/

The resulting output from any API call will give a response like the one below *(the details are not the same, but it is an example)*. Where the user is given a JSON file with the following keys with the most relevant content from the given query.

- url
- date_of_publication
- headline
- main_text
- JSON array of the locations that were included in the report
- diseases
- syndromes

```
{
  "url": "https://www.who.int/csr/don/17-january-2020-novel-coronavirus-japan-ex-china/en/",
  "date_of_publication": "2020-01-17 xx:xx:xx",
  "headline": "Novel Coronavirus – Japan (ex-China)",
  "main_text": "On 15 January 2020, the Min
istry of Health, Labour and Welfare, Japan (MHLW) reported an imported case of laboratory-confirmed 2019-novel coronavirus
(2019-nCoV) from Wuhan, Hubei Province, China. The case-patient is male, between the age of 30-39 years, living in Japan.
The case-patient travelled to Wuhan, China in late December and developed fever on 3 January 2020 while staying in Wuhan.
He did not visit the Huanan Seafood Wholesale Market or any other live animal markets in Wuhan. He has indicated that he
was in close contact with a person with pneumonia. On 6 January, he traveled back to Japan and tested negative for influenza
when he visited a local clinic on the same day.",
  "reports": [
    {
      "event_date": "2020-01-03 xx:xx:xx to 2020-01-15",
      "locations": [
        {
          "country": "China",
          "location": "Wuhan, Hubei Province"
        },
        {
          "country": "Japan",
          "location": ""
        }
      ],
      "diseases": [
        "2019-nCoV"
      ],
      "syndromes": [
        "Fever of unknown Origin"
      ]
    }
  ]
}
```


4.2.5 Handling Errors

To indicate whether our API has responded successfully, we will use the standard response status codes in the HTTP protocol. When input is valid and a JSON response is created successfully, we will pass the **200 OK HTTP response status code**. If the user has provided invalid input, we will return a **4xx** status code to indicate a client error. If the API service runs into an internal error, we will return a **5xx** status code to indicate a server error.

The following table shows what status codes will be returned for specific scenarios. The status code will be returned in the meta-information of the response, and the error message will be returned in the response using this format:

```
{
        “errorMessage”: <ERRORMESSAGE>
}
```

HTTP Response Status Code	Scenario	Example Error Message
400 BAD REQUEST	- When the request is invalid, e.g. invalid or missing timestamp	400 ‘key_terms’ must not be empty

401 UNAUTHORIZED	- When the user does not provide a valid API key	401 Please provide a valid API key
404 NOT FOUND	- When the endpoint cannot be found on our API service	404 '/foo/bar' does not exist
500 INTERNAL SERVER ERROR	- When an uncaught error/exception is thrown in our API Service, e.g. null pointer exceptions	500 Internal server error
501 NOT IMPLEMENTED	- When an endpoint under development is called. This error will not be thrown in our final product.	501 '/foo/bar' is still in development
502 BAD GATEWAY	- When Cloud Firestore returns an invalid response (or a response the our API service cannot interpret)	502 Server returned invalid response
504 GATEWAY TIMEOUT	- When Cloud Firestore is unavailable - When Cloud Firestore is taking too long to respond	504 Server did not respond in time

Note: If no results are found in a search query, we will return an empty list with the **200 OK HTTP response status code**.

The 200 response code is to show that the query has not malformed, that the request was received and the response is empty. An example would be that a user gets an empty array because no results matched their criteria, an error code like 404 would've suggested a wrong endpoint or otherwise, which is not the intention nor what happened.

4.3 Present and justify implementation language, development and deployment environment (e.g. Linux, Windows) and specific libraries that you plan to use.

4.3.1 Overview

Our team decided to implement the web scraper in python along with the beautiful soup library. We will be developing our API in a serverless fashion with the help with cloud firestore. Our data will be populating into the firestore as our database. We will have an API to establish connections between clients and database. It will be documented with stoplight, and having its endpoints built by cloud

functions. Those cloud functions will upload the collected data to firestore, as well as respond to clients through the frontend. Frontend will be built using React framework, CSS and HTML5.
Our design choices were justified by comparing with other possible tools that can be used during development. Those justifications will be thoroughly presented below.

4.3.2 Backend Language

We finalised our decision on Python as our scrapper + API language. This is because python is suitable for developing small applications and prototyping, and all of our members have experience in Python. Since other languages do not provide any outstanding benefits to reduce development time, we decided to use Python. Our API will be created and maintained using python along with cloud functions.

Python	Java	Conclusion
Python is a **scripting language**. This means it is: 🟢 more suitable for developing small applications 🟢 extremely suitable for prototyping in the early stages of application development 🔴 harder to develop additional features once MVP has been made unless OOP has been used	Java is a **verbose and compiled language that enforces Object Oriented Programming**. This means it is: 🟢 more suitable for creating stable products 🟢 easier to build additional features on top of MVP since OOP encourages future proofing through its design patterns 🔴 requires more time to develop an MVP 🔴 requires the developer to understand OOP principles	Due to the small time frame of our development (8 weeks), we would want to use **Python** to quickly develop a MVP.
🟢 All of the developers on our team already know Python	🟡 Only two developers on our team know Java, and only at a beginner level. However, we are open to learning a new language if it results in a better product.	We would want to use **Python** if Java does not provide any outstanding benefits to reduce development time.

		We need to complete the API before D2. The time is less than 3 weeks, and we cannot afford the cost of learning Java.

4.3.3 Scraper

Since the team decided to use Python as our scraper language, we included the above modules into our discussion of finding the most appropriate library to use when scraping the web. Most of our group members do not have any experience in web scraping therefore we wanted to choose a tool that is easy to start.

We finalised our decision on BeautifulSoup4. BeautifulSoup4 is a beginner-friendly language that has extensive documentation. Selenium as another beginner-friendly language is considered a good choice. The advantage of it is that it can run Javascript, however in this case it is not necessarily needed since the content is loaded statically from the base url. Also using Selenium needs a browser running which occupies more resources and makes it slower.

Scraping tools	Easy of use	Documentation	Speed
Beautiful Soup	🟢Beginner-friendly	🟢Extensive documentation	🔴 slow
LXML	🔴 not beginner-friendly	🔴 Documentation is not beginner-friendly	🟢 fast
Scrapy	🔴 not beginner-friendly	🟢Excellent documentation	🟢 fast
Selenium	🟢Beginner-friendly	🟢Extensive documentation	🔴slow
MechanicalSoup	🔴 not beginner-friendly	🔴unmaintained for several years, does not support python3	🟡medium

Urllib	🔴 not beginner-friendly, bad interface	🔴 Documentation is not as easy as other choices	🟡medium

4.3.4 Deployment

Our API will be deployed serverlessly by using Cloud Firestore. We decided to choose a serverless backend over a server-based one because a serverless backend has an easy setup and we won't have to run the server constantly. A serverless backend also eliminates the concern towards operating systems. In between two different serverless services - Cloud Firestore and AWS Amplify, we decided to choose Cloud Firestore since google cloud's free plan is more suitable for this project and the functionalities it provided will suffice.

Hence we chose google cloud functions to support flasks original functionality, which allows us to host our flask functions so that our API can run in a serverless state.

Cloud Firestore (serverless)	**AWS Amplify (serverless)**	**Server-based backend(eg. Python flask)**	**Conclusion**
🟢Do not need to run the server constantly	🟢Do not need to run the server constantly	🟢Have full control to our server 🟢Free to choose team's preferred language 🔴Have to run the server constantly	No preference
🟢 Firestore is very easy to set up, while being very user friendly and easy to use with python.	🔴Higher learning costs, because AWS has many modules to deal with different problems. We need to learn the corresponding modules for each requirement	🟢Members have some flask+python experience 🔴But if you need to deal with different problems, you may even need to have a deeper understanding of related libraries	In terms of learning, firestore is almost the best choice. Because he can let us develop our project quickly

🟡Due to the serverless structure, there may be some delays when a "cold start" occurs, but in most cases it is okay. Refer here for cold starts definition	🟢The optimized database provides better performance.	🔴This is based on the servers we can rent, but in fact, due to insufficient scalability, it is likely to exceed the access limit.	AWS may be the best choice, but Cloud Firestore is close behind.
🟢Excellent synergy brought by serverless architecture. This reduces the reliance on hardware and operating system	🟢Excellent synergy brought by serverless architecture. This reduces the reliance on hardware and operating system	🔴Need to depend on the operating system.	At this point, AWS and Cloud Firestore have gained significant advantages.
🟢Has a 90 days free trial/$300 credit with unlimited access to all functionalities. If we go over time/over budget, we need to pay a fixed amount of money regularly.	🟢Free trial for 750 hours, however team members might accidentally leave an instance on and thus use up a big chunk of our time allowance.If we go over time, we need to pay a fixed amount of money regularly.	🔴We need to determine the load of the server in advance, and rent a suitable server, which may be very expensive. The fee depends on the usage of the server instead of paying a fixed amount regularly.	Cloud Firestore is one point ahead in the price war, because we cannot estimate whether our current load will exceed its free limit, so we might as well try Cloud Firestore first, and it is cheaper than AWS.

External database's hosted by us were not a viable option as they they do not provide the scalability, reliability and ease of use that we require, due to the large amounts of data we will be storing. As such we decided to go cloud.

4.3.4.0.1 Cold Starts

Cold starts can be defined as the set-up time required to get a serverless application's environment up and running when it is invoked for the first time within a defined period. Cold starts are somewhat of an inherent problem with the serverless model

4.3.5 Database

We are going to use firestore as our database instead of realtime database. This is because querying will be easier and it provides offline web support.

	Realtime Database	Firestore	Conclusion
Offline support	🔴 Offline support only for mobile (Android & iOS)	🟢 offline support for both mobile and web clients.	We tend to choose Firestore since the key point is that we need support for **web** clients.
Data model	Realtime Database is a giant JSON tree. 🔴 Complex and hierarchy based data is hard or organized when it's scaled. 🔴 Creating a query across multiple fields is hard and might denormalize data	In Firestore, data is stored in objects called documents that consist of key-value pairs. 🟢all queries are shallow, makes fetching data cleaner since we don't have to fetch all of the linked subcollections.	Firestore is better in this case. A better Data Model makes it easier for us to build a database. There is a lot of data for this project, and we also need to provide various auxiliary data to support the subsequent prediction function (if it has one).
Querying	Support in-depth query. 🔴can use filters or sorting on the attributes in the query, but cannot handle both at the same time.	You can use index queries with good composite filtering and sorting. 🟢You can use sorting, combination filtering and chain filtering for each attribute in one query.	We choose firestore. Supporting both filters and sorting in a query is critical. Imagine that when you query the *Asia-Pacific region* with the *most COVID-19* in *January 2021*, firestore will have excellent performance.
Scalability	🔴The single-region solution needs to be expanded by itself.	🟢Multi-regional solution without self-expansion.	Cloud Firestore is ideal in this case because it reduces any potential maintenance pressure in the future.

4.3.6 Frontend

The frontend of our program will be constructed using the React framework, CSS and HTML5, because it is easy to learn and many of us have experience.

React	Vue	Conclusion
🟢Used by a large number of companies, and has a wide range of front-end development libraries. 🟢Some members have used React, so the learning cost may be lower.	🟢The same powerful front-end development framework is also an industry standard. 🔴No one in the group has used it. Although the cost of learning is not high, it is better to choose React that already has experience in using it.	We will be using React. Both are powerful tools for frontend construction however our team members have more experience in React.

4.3.7 Development

For local development, the API and scraper will run on windows during the initial development, and use the virtual machine (VMware) provided by the school to test their performance under Linux.

	Windows	Linux	Conclusion
Development environment	🟢Everyone installed windows	🔴Can only be developed on the school's vlab	This tends to be windows
Operating environment **(Compatibility)**	🟡Need to use a virtual machine to determine whether it can run on the school's Linux system. 🟢There are some tutorials on the Internet	🟡Can run smoothly in school. But it needs to be tested in the windows environment.	We can use the school's vlab for testing Linux and use **VMware** test other environments, so it's no harm at all.

	that can use **Hyper-V or VMware**		
Team experience	All team members have experience		

4.3.8 VII. API documentation

We are going to use stoplight over swagger. StopLight allows multiple people editing at the same time therefore lifts the pressure of having one person documenting the whole API. Stoplight also provides sufficient functionalities for API design and documentation. Therefore we chose StopLight over Swagger.

StopLight	Swagger	Conclusion
Free plan allows for 5 members in a team editing at the same time. 🟢free 🟢write documentation more freely without relying on only one team member	Does not allow multiple users editing at the same time unless the team upgrade the plan which costs money 🔴 not free 🔴 heavily dependent on one team member	We would prefer **StopLight** in this case.
Stoplight has a clean interface , which makes it easy to navigate. 🟢easy to navigate.	Swagger has a clean interface especially when integrating with React applications. 🟢easier for frontend development since swagger and React can be easily integrated.	We would choose **Swagger** in this case since we are going to use React for frontend.

4.3.9 Testing

We also had a discussion on which Python testing framework to use. Below is a comparison between PyTest and UnitTest.

PyTest	UnitTest	Conclusion

🟢 PyTest is a reputable third-party testing library	🟢 UnitTest is a standard unit test framework module in Python	We can use **either testing frameworks**.
🟢 PyTest reduces boilerplate code so that the testing logic is more apparent	🔴 UnitTest is more verbose which can obscure the testing logic	We would want to use **PyTest** to help write tests easier and faster.
🟢 PyTest has more plugins than UnitTest	🔴 UnitTest has less plugins than PyTest	We would want to use **PyTest** to take advantage of its flexibility.
🟢 PyTest is compatible with UnitTest	🔴 UnitTest is not compatible with PyTest	We would want to use **PyTest** to reduce compatibility issues in the future if we were to use UnitTest
🟢 All developers in the team have used PyTest before	🔴 No developers in the team have used UnitTest before	We would want to use **PyTest** to reduce time learning a new framework and to focus on writing tests.

Ultimately, we chose PyTest mainly because we were all familiar with the framework, and it also provided more flexibility than UnitTest.

4.4 Management Information

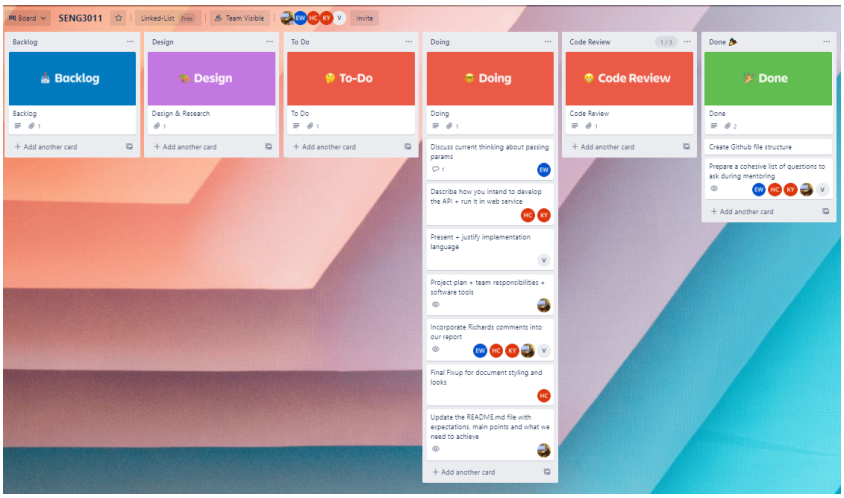
4.4.1 Team Member Responsibilities

Each team member has their own delegated responsibilities, which will mostly fall under their roles - shown below.

Member	**zID**	**Role**
Thomas Sinn	z5213546	Project organisation, Scrum master, frontend developer
Hayes Choy	z5258816	Architecture, Backend Lead
Eddy Wong	z5207607	Overall Documentation, backend developer
Leila Yuan	z5261559	API, backend developer, Stoplight documentation
Xiyang Shi	z5137765	API, Frontend Lead

Phase one included completing the skeleton of the Github repository and placing the respective documents within it. Since there are many tasks for this phase we just delegated per task, at the same time, we stuck to our strengths. One person focused on testing, one on API implementation, one on research, design and documentation, two on data collection. However, with regular standups, we are able to shift our focus to any part of the group, and help out when necessary.

These responsibilities will appear on our team's trello board https://trello.com/b/f0MnxHBc/seng3011, sprint 1 is shown below, clearly assigned to each team member.



- Backlog refers to the work that could be done in the future
- Design comes even before we start developing, we are trying to shape the product to fit the requirements
- To-Do is the active tasks that need to be completed within the current sprint
- Doing is the current tasks that have been allocated to individuals and are currently being worked on
- Code Review, is they are being checked and questions are being asked about their validity.
- Done is a complete task that has been approved and no longer needs to be worked on.

4.4.2 Work Arrangements

Stand ups

Tuesday 9pm, Thursday 2-3pm or 7-8pm on Discord

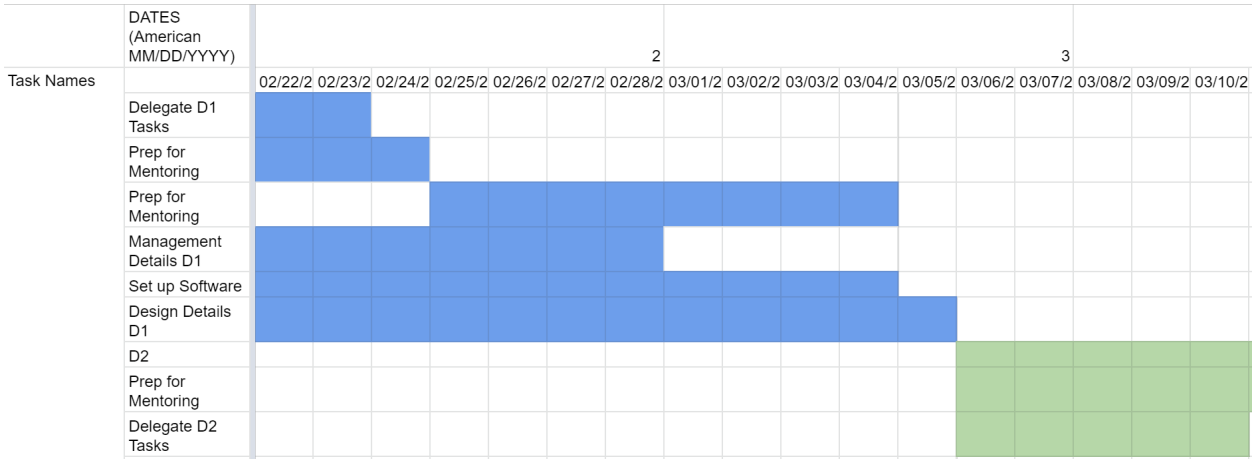
Standups will be led by the scrum master, where each team member will quickly say what they have completed since last standup. Then the team will review and see if all the other team members are meeting deadlines and staying on task by all looking at the gantt chart. In addition to this, this is a time to ask questions and ask for help doing your individual tasks, especially if you have blockers. After figuring out where each team member is at, we can plan the next sprint accordingly.

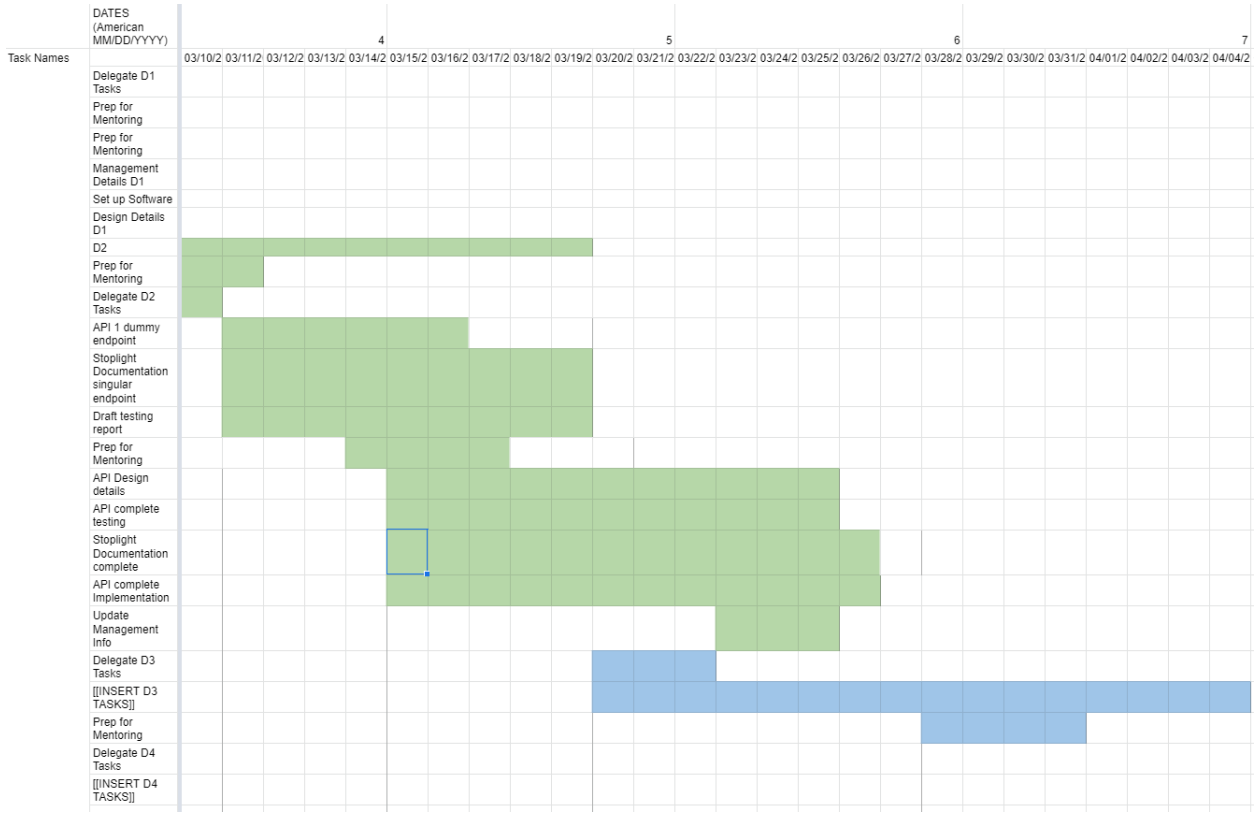
Mentoring

Thursday 7.40-8pm on Discord

Mentoring will consist of updating one of our stakeholders, on the progress of our project. Before mentoring, we will have a comprehensive list of questions to ask the stakeholder. After the meeting, this is when the scrum master will delegate new and iterated tasks.

Gantt Chart Snapshot: Sprint 1 and start of 2



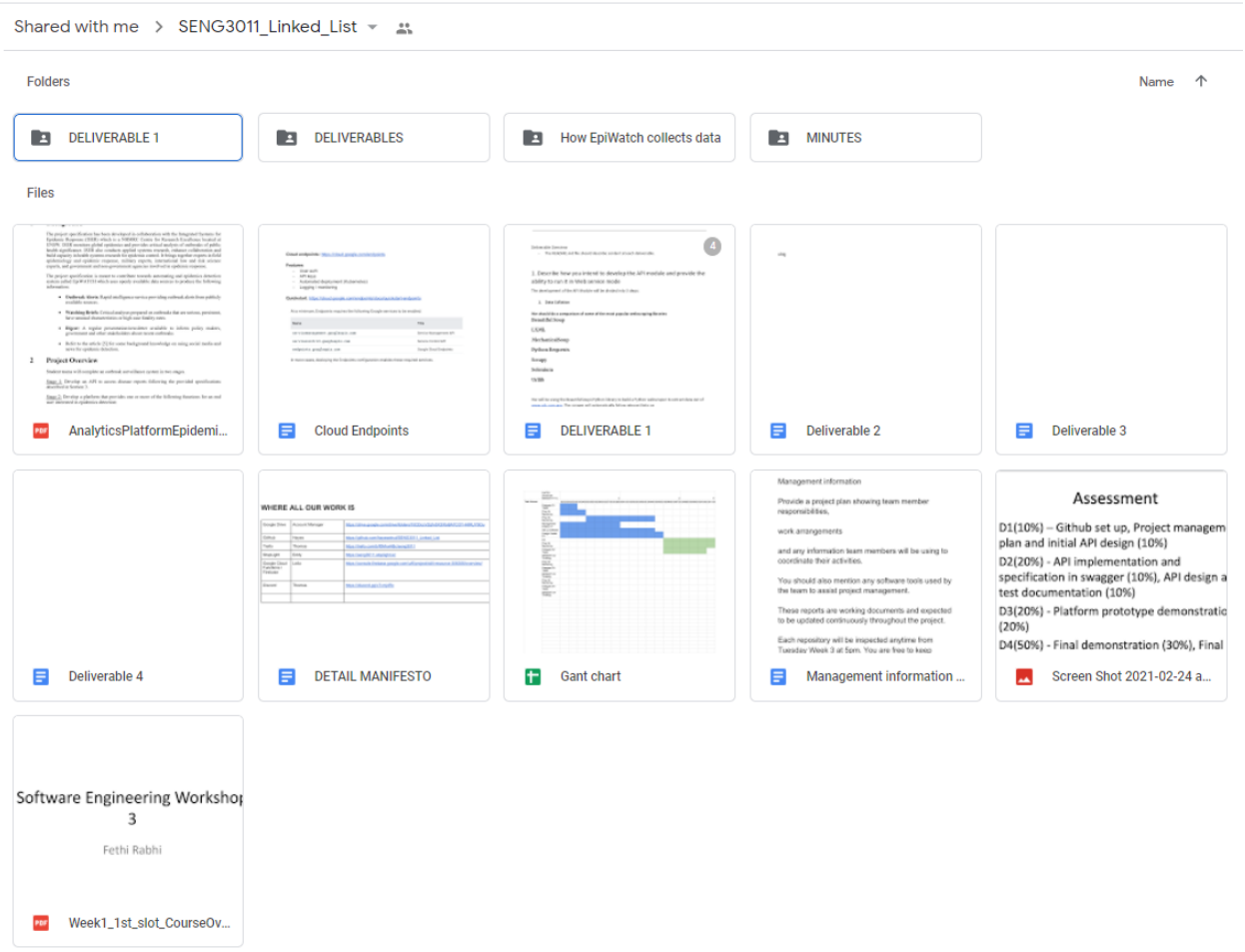


4.4.3 Software Tools

Our team uses a variety of software tools to keep on track as well as work well as a team.

Google Drive
https://drive.google.com/drive/folders/1ltCDccVZqfn5X3XfyBA7COY-HtRUY9Ou

Google drive is a great software tool for our team because it keeps all of our documents in a central location. In addition to this, it allows the whole team to edit and collaborate on documents at the same time.

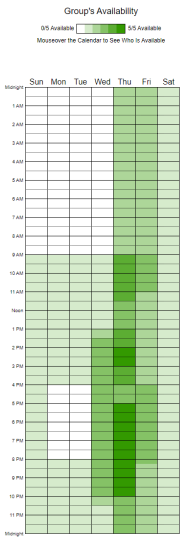


Trello (Image above)
https://trello.com/b/f0MnxHBc/seng3011

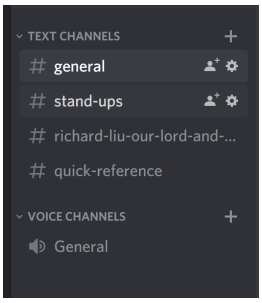
Trello is a management tool that we use to keep on task, and know what tasks are delegated to us, as well as see what other team mates are doing. It also allows us to plan our sprints and meet sprint deadlines.

When2meet (see right)
https://www.when2meet.com/?11135331-QjeoB

When2meet is a very simple tool that we use to plan around peoples very busy schedules. Since our team has multiple people who work part-time jobs, it is essential that we have a place to easily see when people are free to pair and set up one-on-ones.



Discord (see right)
https://discord.gg/c7vmj4Rc



Discord is where all of our meetings, sprint, one-on-one, mentoring etc are done. This is also where we communicate asynchronously and keep people up to date with what we are doing, and try and get rid of blockers as soon as they appear.

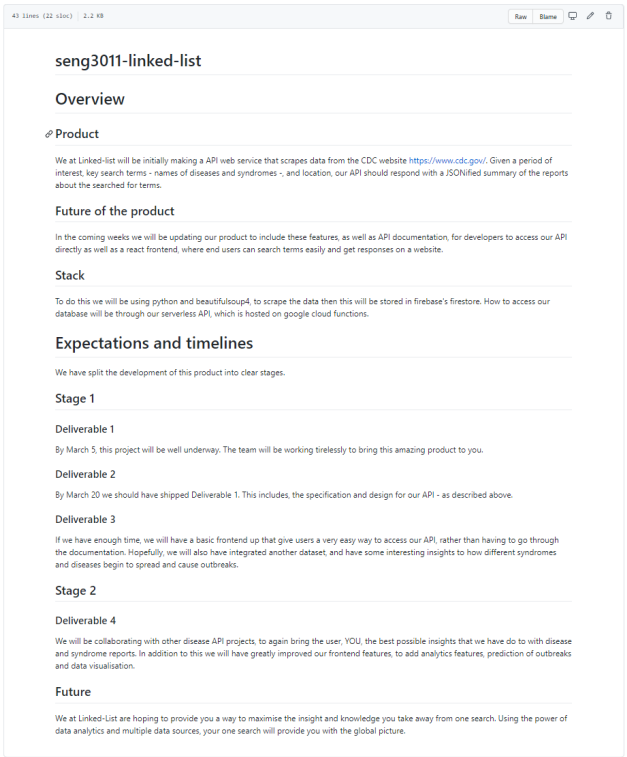
Gantt chart (see above: gantt chart snapshot)

https://docs.google.com/spreadsheets/d/1-gy4oQcdr1Q5BSLCZolsHpIAhTh--26X7evYE7ma3_Y/edit?usp=sharing

A Gantt chart serves as our team's roadmap. This roadmap will take into account engineering constraints, as well as deadlines and deployment dates, and give us a big picture view on what is happening with our product beyond the current sprint. This gives us context to deeply understand the requirement of the epics that are within this sprint and how they fit into the overall product.

Github (see right)
https://github.com/hayeselnut/SENG3011_Linked_List

Github is a software development and version control platform that uses git. Our team will be using github for that exact reason. When we code as a team, without some sort of version control system, the product that we ship to the end user may have issues because we do not know what version of the code is production ready. Git and Github give us that power. This is also where people will be able to follow our teams progress in the README.md, and use our open source code.


43 lines (22 sloc) 2.2 KB Raw Blame

seng3011-linked-list

Overview

Product

We at Linked-list will be initially making a API web service that scrapes data from the CDC website https://www.cdc.gov/. Given a period of interest, key search terms - names of diseases and syndromes -, and location, our API should respond with a JSONified summary of the reports about the searched for terms.

Future of the product

In the coming weeks we will be updating our product to include these features, as well as API documentation, for developers to access our API directly as well as a react frontend, where end users can search terms easily and get responses on a website.

Stack

To do this we will be using python and beautifulsoup4, to scrape the data then this will be stored in firebase's firestore. How to access our database will be through our serverless API, which is hosted on google cloud functions.

Expectations and timelines

We have split the development of this product into clear stages.

Stage 1

Deliverable 1

By March 5, this project will be well underway. The team will be working tirelessly to bring this amazing product to you.

Deliverable 2

By March 20 we should have shipped Deliverable 1. This includes, the specification and design for our API - as described above.

Deliverable 3

If we have enough time, we will have a basic frontend up that give users a very easy way to access our API, rather than having to go through the documentation. Hopefully, we will also have integrated another dataset, and have some interesting insights to how different syndromes and diseases begin to spread and cause outbreaks.

Stage 2

Deliverable 4

We will be collaborating with other disease API projects, to again bring the user, YOU, the best possible insights that we have do to with disease and syndrome reports. In addition to this we will have greatly improved our frontend features, to add analytics features, prediction of outbreaks and data visualisation.

Future

We at Linked-List are hoping to provide you a way to maximise the insight and knowledge you take away from one search. Using the power of data analytics and multiple data sources, your one search will provide you with the global picture.

VSCode Live Share

VSCode Live Share is a way of pair programming from different computers. Instead of git where you each have a different local version of the file, this tool allows us to code on the same document at the same time. Therefore, allowing us to debug and ship features faster.