SENG3011 Deliverable 4 - Final Design Report

Group: FatDonkey

Members: Flynn Zhang, Edwin Tang, Frank Su, Josh Rozario

About Us

Due to unprecedented events from the past year, our team has been tasked by ISER (Integrated Systems for Epidemic Response) to develop an outbreak surveillance system to further assist their Epiwatch software which uses public data sources to detect outbreaks.

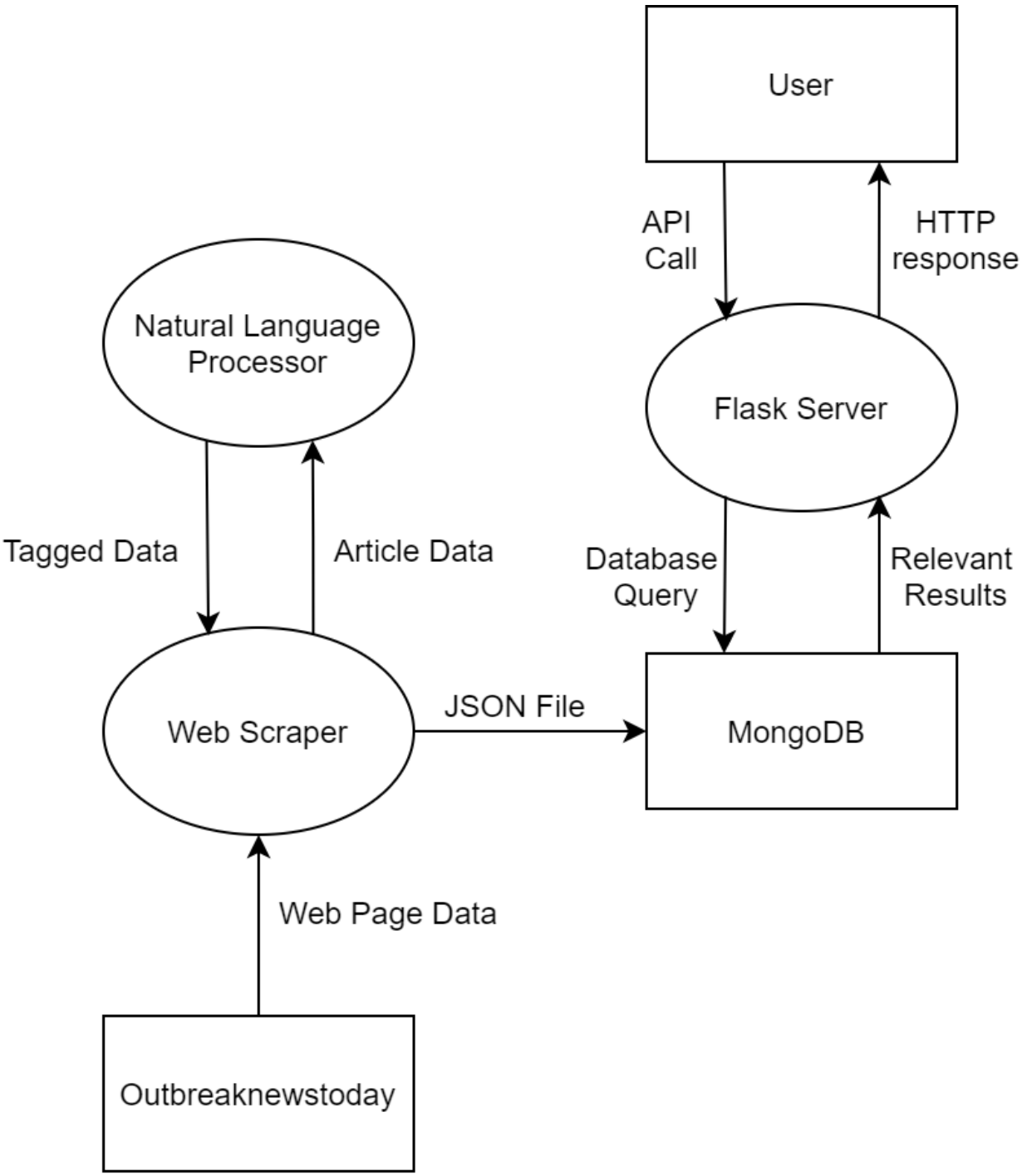
The first half of our report details stage 1 of our project where we developed an API which retrieves disease reports from a data source (outbreaknewstoday.com) that are relevant to the parameters parsed to it. Our rationale for programming language, hosting, storage and framework decisions will be included.

In stage 2 of our project, we developed a web application called TravelGuard, a travel advisory application designed to keep users up to date with COVID statistics, other disease outbreak information and travel recommendations. This application made use of our API from stage 1 as well as other APIs.

Stage 1 System Overview

Initially, the articles from the site will be recursively scraped and stored. After, all articles will be passed to a natural language processor to extract the name of the disease and the location of the disease and add that information to each article element. The tagged data will then be stored in a database for easy retrieval from our API. A user will be able to call our web server after it has been deployed to request their desired data. In order to maintain an updated database, we will scrape and tag articles once a day.

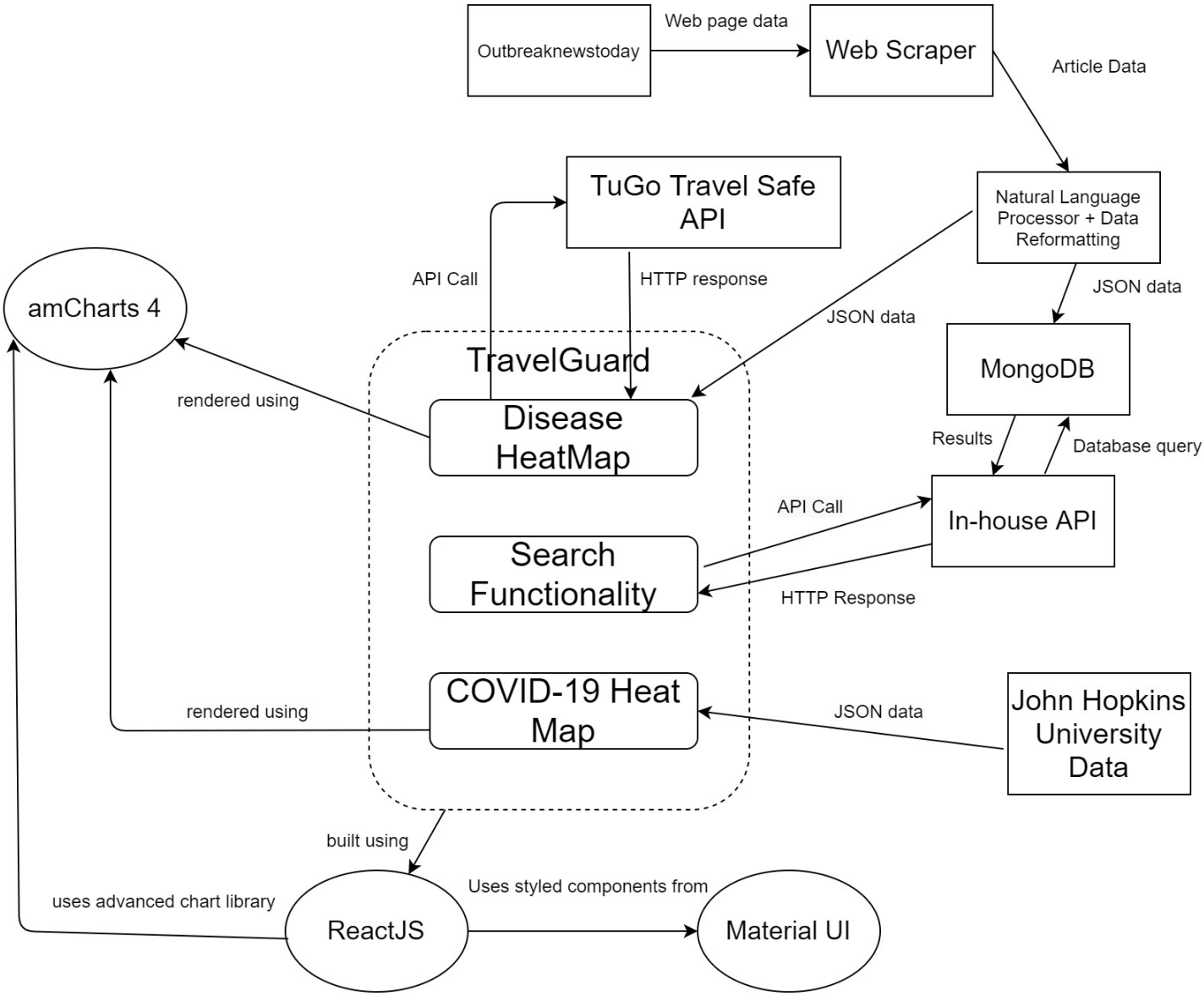
The following diagram is a visual depiction of how our system will operate:


User
API
Call
HTTP
response
Natural Language
Processor
Flask Server
Tagged Data
Article Data
Database
Query
Relevant
Results
Web Scraper
JSON File
MongoDB
Web Page Data
Outbreaknewstoday

Stage 2 System Overview

Our web application TravelGuard was built using the Javascript framework, ReactJS. Our graphs were rendered using the JS data visualisation library, amCharts 4. Travel advisories were provided by the TuGo Travel Safe API and disease data was sourced from our in-house API developed in stage 1 which scraped and processed articles from outbreaknewstoday.com. Our COVID-19 heatmap used data from John Hopkins university and we used styled components from Material UI to provide a custom theme for our website.

The following diagram is a visual depiction of the technical architecture of our web app.



API Use Case

Use Case 1:

As a developer I want to fetch any news articles about any disease outbreaks from outbreaknewstoday.com so that I can provide gather disease information to display in my application

Main use case:

A developer creates a HTTP request with the name of their disease in the disease parameter of the URL. The URL is https://disease-watcher.herokuapp.com. Once the API receives the request it which will return any disease articles that refer to an outbreak of the the input disease

Alternative Flow 1:

The user can also further input a location or region in the respective parameters of the URL and the API will only return articles that refer to the input location or region.

Alternative Flow 2:

The user can also input a start_date and an end_date in the respective parameters of the URL and the API will return articles that were published between the two dates.

Alternative Flow 3:

The user can also input a start_index and end_index in the respective parameters of the URL and the API will return articles that are listed between the two indices

TravelGuard Use Case

Use Case 1

As a traveller I want to view disease outbreaks by country so that I am aware of the diseases that I may encounter when i travel to that country.

Main use case

A user loads the TravelGuard HomePage. They can then scroll the date slider on the main chart which will show the outbreaks as orange bubbles on their geographic location. The user can hover over the bubble to see the name of the disease. The user can then click on the orange bubble to view more details about the reported outbreak on the sidebar.

Alternative Flow 1

If the user wishes to view the original report they can click on the link to the article at the bottom of the report.

Use Case 2

As a traveller I want to see up to date information on the diseases I may encounter in the country so that I can take precautionary measures before visiting the country.

Main use case

A user loads the TravelGuard HomePage and clicks on the country they wish to view disease information about. They can then see a list person-to-person and food/water related diseases for that country.

Alternative Flow 1

If the user wishes to view more details about the disease they can click on the relevant disease and a modal will pop up, revealing how a traveller would mitigate the likelihood of being exposed or contracting this disease. It also explains where the user might be exposed to the disease in the country.

Use Case 3

As a traveller I want to see a list of recommended vaccines that I should take before travelling to a certain country so that I can significantly reduce the risk of being sick on the trip.

Main use case

A user loads the TravelGuard HomePage and clicks on the country they wish to view vaccine information about. They can then see a list of recommended vaccines for that country.

Alternative Flow 1

If the user wishes to view more details about the vaccine they can click on the relevant vaccine and a modal will pop up, revealing a modal which details what situations a traveller might be exposed to the virus and how likely they will need the vaccination.

Use Case 4

As a traveller I want to know the quality of healthcare in a country so that I can know what kind of assistance i may receive in the event of getting sick.

Main use case

A user loads the TravelGuard HomePage and clicks on the country they wish to view healthcare information about. They can the click on “Medical services and facilities” to view a qualitative description of healthcare in the country and how much it might cost them to receive it.

Use Case 5

As a traveller I want to know how many COVID-19 cases a particular country has so that I can assess how dangerous it is to travel to that country.

Main use case

A user loads the TravelGuard HomePage and clicks on the “COVID-19” nav item. The covid-19 heatmap is then displayed and the user can hover over a country of their choice to see how many cases the country has.

Alternative Flow 1

A user can click on the country to change the line graph at the bottom of the page to display the covid data for the specific country. The line graph displays the historical number of COVID cases for each country and can tell the user whether the country is experiencing a spike in cases or if the country is recovering and is in a downward trend.

Use Case 6

As a traveller I want to compare the number of COVID-19 cases between different countries so that I know which countries are safer to travel to.

Main use case
A user loads the TravelGuard HomePage and clicks on the “COVID-19” nav item. The covid-19 heatmap is then displayed and the user clicks on the slider at the top of the chart to toggle to “Per Capita” mode. In this mode, the heatchart colours countries with higher cases per capita with a brighter orange and countries with lower cases per capita with a darker orange. From this feature, the user can tell which countries have higher cases per capita.
Alternative Flow 1
A user can hover over a country to see how many cases per million a country has.
Alternative Flow 2
If the user toggles back to “Absolute” mode, the user can see orange bubbles on the heatmap which are scaled relative to the number of cases the country has. The user is able to immediately see which country’s bubbles are bigger and hence which countries have higher numbers of cases.

Use Case 7
As a traveller I want to see the recovery and death numbers in a country so that I know my chances of surviving a covid infection.
Main use case
A user loads the TravelGuard HomePage and clicks on the “COVID-19” nav item. The covid-19 heatmap is then displayed and the user clicks on the “recovered” bubble in the bottom right of the heatmap. The chart will display the recovered cases instead of active cases.
Alternative Flow 1
A user can click on the “deaths” bubble to see how many deaths each country has.
Alternative Flow 2
A user can toggle to “per capita” to see recoveries/deaths per million for each country.

Use case 8
As a traveller I want to see all disease reports for a certain country so that I am aware of all the historical outbreaks the country has had.
Main use case
A user loads the TravelGuard HomePage and clicks on the “Search” nav item. The user types in a country in the country field and clicks search. The table below displays the title, disease, date and body of each report.
Alternative Flow 1
A user can search for reports that only occurred after or before a certain date as well reports that occurred between 2 dates by inputting the “Outbreak reported after” and “Outbreak reported before” fields.

Use case 9
As a traveller I want to see all countries that a disease has appeared in so that I am aware of the geographic regions that the disease has appeared in.

Main use case
A user loads the TravelGuard HomePage and clicks on the “Search” nav item. The user types in a disease in the disease field and clicks search. The table below displays the title, country, date and body of each report.

Use case 10
As a traveller I want to learn about the different types of infections so that I know how to treat myself or fellow travellers if we get sick on the trip.
Main use case
A user loads the TravelGuard HomePage and clicks on the “Learn” nav item. The user can see the 3 common types of infections and an information box detailing what drugs/medicines work against each type and how they may spread.

API Module
We used Python and Flask to develop our Restful API and scraper as we have all used the framework and language in the past.

Scraping the data
The scraping of our data source http://outbreaknewstoday.com/ will be carried out using Scrapy, a popular Python module that our team also has experience with. It will extract the date,url, region and title of the article as well as the body of news reports. Extracting the metadata will help categorise the reports and help users of our service refine their search. We used CSS selectors to extract information from the HTML elements which contained our desired data. E.g. **response.css('div.posttitle').css('h1::text').get(),** The website had around 5-6 links to related articles at the bottom of the page so our crawler would open and scrape each related article. Then the scraper would repeat the process and scrape the links at the bottom of each of the previously mentioned pages. This process would repeat many times. Due to the recursive nature of this scraping, our scraper was able to scrape around 8000 articles in a few minutes, some articles scraped were written as early as 5 years ago. The scraper would store all the scraped results as a json file.

Cleaning/tagging the data
We believed there would be certain challenges in extracting the location and disease name metadata as this information is embedded within the title of the news report.
To solve this problem, we will use text analysis AI (Spacy) to extract the desired data. Spacy is a python library that conducts natural language processing (NLP). We had originally planned to use another library called NLTK but it’s accuracy in extracting our desired data was not up to our standards. We needed Spacy to extract 2 critical pieces of metadata which were the **location** of the reported outbreak and the **name of the disease** that was reported. These 2 tasks fell under a subcategory of NLP called **Named Entity Recognition**,

essentially the identification of proper nouns. Named Entity recognition works by using a model supplied by the user to quickly label words in a sentence.
For example,

In this sentence, the model was able to correctly tag the word “Apple” with ORG meaning that it recognised Apple as the name of tech company and “UK” as a geographical location. (GPE tag).

We used the default **en_core_web_sm** model to identify the names of countries or states. Our tagger would look for the first word in the title of the article with the “GPE” tag and set that as the location of the outbreak. If it was not able to detect a location in the title, it would try the first sentence of the body of the article.

Identifying the location of the outbreak was the easier of the two tasks as it is a very common feature that programmers might use. As a result, the accuracy of the tagging was quite good. Identifying the name of the disease was much harder as it is quite a niche area in named entity recognition. However, we were able to find a Python package called **scispacy** which had models specifically related to identifying scientific words. In particular there was a model called **en_ner_bc5cdr_md** which was built for identifying diseases and chemicals. We used this model to identify the name of diseases and it still had some trouble identifying diseases so we applied the earlier strategy of looking for a disease in the title first, then the first line of the body if it was not able to find it in the title and then in the second line of the body as a last resort.

With the use of spacy and scispacy we were able to tag 70% of articles with a location and disease with about 2500 unique articles not having either a location or a disease tag. 700 articles did not have a location nor a disease tag.

<u>The database</u>
We imported the tagged data json file into MongoDB, our NoSQL database management system which was perfect for processing our document based data (json files). Using a DBMS allowed the API to efficiently query this database without having to access each individual element inside a json file whenever a call was executed. From there onwards the python driver Pymongo provided by MongoDB allowed us to modify the queries to the database similar to how one could modify a SQL query to retrieve specific results. This served as a way to process and retrieve the relevant articles in the database as per the user inputs passed to our API endpoint.

<u>Webserver</u>

Our API was built using Flask, a lightweight web server framework. When a call is made to our flask server, the arguments of the query string (if any) are parsed and checked for their validity. If any of the inputs are invalid, the server would respond with a 400 error code and an appropriate error message. If the inputs are all valid, the server would search the database based on the given parameters and return the results .

We built individual functions for each parameter that could be given to the server. For example, there are separate functions for filtering based on location, disease name, etc. The main function would just call any combination of these functions to obtain the desired data. This allowed concurrent development of the functions and introduced modularity to the design of our server. This proved highly useful during the implementation of our API as it meant that multiple called functions serving different roles that aided the main function could be worked on at the same time, which drastically decreased production time and led to our timely implementation of our system. The modularity of our implementation also allowed us to test each function in isolation as to remove any unnecessary variables during testing that could cause errors. As such we were better able to isolate any errors allowing us to fix them more easily. If the request was successful, the response from the server would be a list of articles in JSON format.

We used the Flask-Restplus extension to automatically document our API endpoint through a Swagger UI. However, many aspects of documentation needed to be manually added. The response codes and messages were added through decorators.

```
@api.response(200, 'Success', article)

    @api.response(400, 'Bad request', error_msg)
    @api.response(404, 'URL not found')
    @api.response(500, 'Internal Server Error')
```

Similarly, the description of the input parameters was also added through decorators.

```
@api.doc(params={'location' :'Country or
State/Province (e.g. China)',
                'disease' : "Type of Disease (e.g.
Cholera)",
                'start date': 'Outbreak reported on
or after this date (dd/mm/yyyy)',
```

```
                'end date': 'Outbreak reported on
or before this date (dd/mm/yyyy)',
                'region': 'Geographic region of
outbreak (e.g. Europe)',
                'results': 'Number of results (Must
be > 0)(e.g. 10)' })
```

Flask-restplus allowed us to add “models” to describe the structure of the response of our API. We were able to model the structure of an individual “article” object and display it on the Swagger UI.

```
article = api.model('article', {
    "title": fields.String(example="SARS, MERS ruled out in China pneumonia cluster"),
    "date": fields.String(example="January 5, 2020"),
    "location": fields.String(example="China"),
    "region": fields.String(example="Asia"),
    "url":
fields.String(example="http://outbreaknewstoday.com/sars-mers-ruled-out-in-china-pneumo
nia-cluster-40965/"),
    "disease": fields.String(example="SARS"),
    "body": fields.List(fields.String(example="Health officials in the city of Wuhan in Hubei
province, China ")),
  } )
```

We were also able to log the requests and responses of the server and write it to two files using decorators which would trigger functions before and after a request. The logs are captured in two files, one that is simpler, easier to read, the other more verbose and contains more detail. Details such as how long the request took to process are recorded.

Deployment to the cloud

We deployed our API to the web using Heroku’s Cloud application platform for its convenience and widespread availability. Deployment using Heroku, required a requirements.txt file which had a list of dependencies that Heroku needed to install for our application to work. Our app was successfully deployed using the Heroku-20 stack for Python applications and can be found at this link: https://disease-watcher.herokuapp.com/

Module Interaction

We developed our API as a RESTful API as it is the de facto standard for creating the architecture of network systems. Our API only has 1 endpoint which is the GET method the user will use to pass certain parameters for the articles that they desire.

Our API takes the following parameters:

'location' :'Country or State/Province (e.g. China)',
'disease' : "Type of Disease (e.g. Cholera)",
'start date': 'Outbreak reported on or after this date (dd/mm/yyyy)',
'end date': 'Outbreak reported on or before this date (dd/mm/yyyy)',
'region': 'Geographic region of outbreak (e.g. Europe)',
'start_index': 'Articles start from this index (Default is 0 which is the first article)',
'end_index': 'Articles end at this index (Default is len(articles) - 1 which is the last article)'

In order to prevent significant latency when using our API, the user is not able to request all articles in the database and can only obtain a subset of articles by supplying the “location”, “region” or “disease parameter. All other fields are optional. The **start_index** and **end_index** parameters allows the user to see a portion of the total results that would be returned if no indices were set. For example, if the user wanted to see the first 5 results they would set the start_index to be 0 and end_index to be 4.

Valid **region** inputs are: 'Africa', 'Asia', 'Australia', 'Canada', 'Europe', 'Indian subcontinent', 'Latin America and the Caribbean', 'Middle East', 'US News'

Our module will find the relevant articles that matches the user’s parameters from the database. The API call will then be processed and return to the user, a list of the relevant articles as a JSON object. JSON objects are used by most servers and clients to exchange information, thus allowing our API to be more compatible with more users and increasing our user market.

If there is an error during the execution of the API request, it will be handled as per HTTP response codes. Our API has the following error codes:

200 - Success

400 - Bad request (inputs are invalid or missing)

404 - URL not found (invalid route)

500 - Internal Server Error

In particular, a 400 response code would be accompanied with an error message, explaining which input was incorrect.

User Interface

A user can easily test our API using the Swagger UI that has been provided at our API URL, by typing in the parameters and clicking the “Execute” button. The UI is also able to provide the URI path and query parameters for developers if they wanted to execute the call in their code.
E.g.

https://disease-watcher.herokuapp.com/outbreak/?location=China&start%20date=20%2F04%2F2019

API Call Examples

curl -X GET "https://disease-watcher.herokuapp.com/outbreak/?location=China" -H "accept: application/json"

Accept: text/html
Accept=Language: en-us, en

curl -X GET
“https://disease-watcher.herokuapp.com/outbreak/?location=China&start%20date=20%2F04%2F2019" -H "accept: application/json"

Accept: text/html
Accept=Language: en-us, en

JSON Return Example

[

 {

  "title": "Gates Foundation commits $10 million to contain the global spread of 2019-nCoV",

"date": "January 27, 2020",

"location": "China",

"region": "US News",

"url": "http://outbreaknewstoday.com/gates-foundation-commits-10-million-to-contain-the-global-spread-of-2019-ncov-42552/",

"disease": "unknown",

"body": [

"The ",

" announced that it is immediately committing $10 million in emergency funds and corresponding technical support to help frontline responders in China and Africa accelerate their efforts to contain the global spread of 2019-nCoV.",

"The foundation is committing $5 million to the 2019-nCoV response in China and is already working with a range of Chinese public and private sector partners to accelerate national and international cooperation in areas of critical need, including efforts to identify and confirm cases, safely isolate and care for patients and accelerate the development of treatments and vaccines.",

"Partners include the National Health Commission and Chinese Center for Disease Control and Prevention, the National Natural Science Foundation of China and various research institutes affiliated with the Chinese Academy of Sciences, Xiamen University and Sinopharm China National Biotec Group.",

"The foundation is also immediately committing $5 million to assist the Africa Centers for Disease Control and Prevention in scaling up public health measures against 2019-nCoV among African Union member states. These measures will include technical support to implement the screening and treatment of suspected cases, laboratory confirmation of 2019-nCoV diagnoses and the safe isolation and care of identified cases."

]

},]

Web application Module

Our web application essentially has 3 main functionalities/features: the disease heatmap, covid heatmap and search functionality.

Disease heatmap

While our covid heatmap was the main feature in our D3 presentation, following the feedback we received after the presentation, our disease heatmap became the focus of our application. Our disease heatmap is very similar to the covid heatmap as it uses the same mapchart and line graph combination but differs in data source and functionality.

The data was obtained in a very similar process to preparing data for our API in stage 1. Article data was scraped from articles on outbreaknewstoday.com and the countries and disease name had to be extracted from each country using a natural language processor (spacy). However, each article needed to be tagged with a country's ISO code since it was a field that the map needed in order for the chart to display an outbreak. Spacy would tag an article with any type of geographic label whether it be a country, state or city. In order to ensure a country was always tagged, we used Pycountry's fuzzy search which would look up the country associated with a given input. For example, if you searched up "Sydney" using the function, it's first result would be Australia. Using this functionality, we ensured that all the location fields of the article data would be a certain country.

The data also had to be recent as possible with minimal gaps between adjacent dates to ensure the usefulness of our heatmap. As a result, we had to rescrape the data and place restrictions on what articles were stored in the json file. The dates of the articles had to be after January 1st 2020. Due to the depth first search nature of the crawler, articles from the 2nd half of the year would not be scraped so we had to create another crawler where the starting article would be one written in September of 2020. The data also had to be reformatted into the following format:
{"date": "2020-03-12", "list": [{"id": "IT", "disease": "COVID-19"}]
(id refers to the country ISO code)

At this point the data was ready for the chart. Each element in the "list" would represent an outbreak and using the ISO country code the chart would know where to display the bubble. In order for the chart to display the name of the disease once a user hovered over the bubble, we had to change the code to display the name of the disease instead of the number of covid cases (which it was doing previously). We also had to remove the bubbles which allowed the user to toggle between "confirmed", "recovered" and "deaths" as this feature was not provided anymore.

The integration of separate features (travel advice and disease report) with the heatmap on this page was carried out using the very useful State Hook in React which would allow components to listen to changes in a "State" variable and trigger rerenders of the component if there were changes. We had 3 state variables on this page which essentially tracked what country, disease and date was currently selected. The mapchart already had a callback function to handle a user

clicking on a country and carrying out changes in the graph so that it focused on that certain country. We simply inserted a line of code (setState) which updated the “country” state variable into this callback function so that it would keep track of the selected country.

Once, a user clicks on a country, the state variable changes and an API call is sent to the TuGo API to retrieve travel advice for the specific country. This TuGo API would return a lot of travel advisory which we could not all display so we had to be selective with the data that was displayed to the user. The update to the state variable would cause the travel advisory text to be updated. We chose to display the food/water and person-to-person advisories as we felt that eating food/drinking water and interacting with other people were unavoidable events when travelling so these advisories were the most useful.
Below is an example of the http response from the TuGo API.

Since retrieving disease reports from our API required the disease name and date of an outbreak, we made use of our other 2 state variables which tracked the currently selected disease and date. There were similar callback functions which handled the date slider being scrolled and the outbreak bubbles being clicked on. We added lines of codes in those functions to update the date and disease state variables accordingly. When a user clicks on an orange outbreak bubble, an API call is sent asking for a disease report that matches the selected country, disease name (attached to the outbreak bubble) and date that is reflected by the position of the slider. Our database was updated using the new data that we scraped and processed for the chart so for every outbreak bubble that is displayed on the chart corresponds to an article in our database. As a result, an API call will be guaranteed to return the correct report from the database to display to the user.

Covid heatmap

The covid heatmap made use of 2 data sets from John Hopkins University (JHU), one that described covid cases worldwide and another that had covid statistics for each country each day. By default the line chart displays the data for the world and when a user clicks on a specific country, the line chart smoothly transitions to displaying the data for the specific country. The country specific dataset was composed of a list of elements in the would have the following format:


```
[{"date":"2020-12-01","list":[{"confirmed":0,"deaths":0,"recovered":0,"id":"TV"},{"confirmed":0,"deaths":0,"recovered":0,"id":"BV"},{"confirmed":0,"deaths":0,"recovered":0,"id":"GI"}]
```


The element would be a dictionary/object with a date field and then a list of the number of deaths and confirmed and recovered cases for each country as identified by the “id” field which contained the country’s ISO code. The number of active cases is not included so this need to be manually calculated by subtracting the number of deaths and recovered from the confirmed cases.

Our code would loop through the country dataset and for each day it would create an outbreak bubble (orange circle) for a country, if the number of active cases was non-zero. It would then add this bubble to a “bubbleSeries” which is what the map chart used to know which bubbles to display. This process would be repeated for the confirmed, recovered and death numbers so that a user can easily switch between the different modes.

When the slider is scrolled, the bubbles are updated with the data from for the new date by looping through each country in the “list” field for that day.

Search functionality

The search functionality was developed as a convenience for the user because they may not want to go through the hassle of looking for an outbreak report on the outbreak map or if they were having trouble locating a country on the map. It is essentially an easy to use interface for our API from stage 1. A user can search by disease name or country or both on this page and a

HTTP request is subsequently sent to our API to fetch any matching reports. A user can additionally specify a date range for the articles if they were interested in specific time periods. The data used in our API is the same as the data used in our disease heatmap so all the reports from the heatmap are able to be found. Once the response from our API is received, the articles are displayed in an easy to view table.

Learn page

The learn page was implemented as a way for users to quickly identify and understand the differences between possible disease vectors.

The 3 models were implemented with the use of node packages(react-three-fibre, drei, gltfjsx) that enabled webgl rendering of GLB/glTF models . The models that we used were obtained from sketchfab, a site dedicated to the sharing and selling of various 3d models. Once obtained the models need to be pre-processed into a format that is suitable for the web and the packages we used. To do this the package ‘gltfjsx’ was used, to turn the models into declarative and re-usable react-three-fibre JSX components. Once converted the we used the rendering packages react-three-fibe & drei to create a “canvas” (which is the rendering plane for the packages) in order to display the models, the camera positioning was handled by gltfjsx and the interactivity was present in react-three in the form of the function “OrbitalControls”.

```
<Canvas style = {{background: '#000d'}}>
  <ambientLight intensity={0.3} />
  <Suspense fallback={null}>
  <spotLight position={[10, 10, 10]} angle={0.15} penumbra={1}  />
  <pointLight position={[-10, -10, -10]}  />
    <Fungi/>
  </Suspense>
  <OrbitControls/>
</Canvas>
```

Implementation Decisions and Rationale

a. Python
    i. All group members have familiarity with Python and so we can minimise the amount of time spent learning the language
    ii. Language is suitable for the scope and agile development of our project
    iii. Existing libraries will streamline development of our program as various functionalities (e.g. Natural language processing) can be easily provided by Python libraries
    iv. De-facto programming language to interact with the web.
b. Flask
    i. All group members have familiarity with this library from previous projects.
    ii. Serves as easy way to process API calls through implementing them as HTTP endpoints
c. Flask-restplus
    i. Python library to automatically document our API as we develop it by providing a Swagger UI.
    ii. Faster than writing out all of the Swagger UI documentation.
d. MongoDB
    i. Simple and effective NoSQL database management system but is still powerful that will allow us to easily store the scraped data.
    ii. Can be easily set up to interact with Python through Python scripts
    iii. NoSQL can provide superior performance in comparison to relation DBMS
    iv. Does not require data to be transformed into a schema set by the database
    v. Existing data stored in JSON file can easily be imported into a MongoDB database
e. PyMongo
    i. Python library to allow interactions with the MongoDB database
f. CSE Vlab
    i. Default development environment for this project as many of the tools going to be used are already available
    ii. Using CSE Vlab ensures consistency in the development environment among all group members
g. Scrapy.py
    i. Python library for web scraping that will allow us to scrape data from our allocated data source
    ii. Familiarity with this library from previous projects will streamline the development of our scraping program
h. Heroku

    i. Our application is planned to be deployed on Heroku as it is more suitable when considering our project scope in comparison to AWS.
    ii. Allows for easy and flexible management of our system
    iii. Uses Ubuntu 20.04 (default version) as the operating system for the web server which will be supported in the long-term (2025)

i. Spacy
    i. Python libraries to help us easily extract the disease names and locations from the title of each data source
    ii. Can experiment with a variety of prediction models to find one that is the best at identifying desired metadata

j. Regex
    i. Regex used to match strings of user input to the articles stored in database

k. Pycountry
    i. python module used to label articles with an ISO country code by querying ISO country standards database.

l. ReactJS
    i. A popular declarative Javascript framework we used to create the UI and frontend logic for our website. Plain Javascript does not scale well when developing websites as it becomes too verbose and makes bug fixing and reasoning with the logic of your code very difficult. React allows developers to create websites made up of components which allows code to be reused and is significantly less verbose.
    ii. React also allows developers to write the HTML, CSS and JS all in one file due to the component based nature of the language. Allows for more more time-efficient development and caused us less headaches.

m. amCharts 4
    i. Extremely powerful data visualisation tool in Javascript that we used to display our heatmaps and line charts.
    ii. Charts are very customisable and dynamic and allows users to interact with the charts through actions such as clicking a button or dragging a slider.
    iii. Have a diverse range of charts such as their radar timeline or bar chart race that extend beyond basic column graphs or pie graphs

n. Material UI
    i. UI framework to help us quickly develop a site with a consistent theme and layout. Provides styled default components.
    ii. Provides extensive documentation on how to use and customize it's components and provides plenty of examples on how to use each different feature.

o. TuGo TravelSafe API

i. Provided travel advice, health advice and government policy for our travel advice page.
ii. Allows users to look up travel advice for any destination
iii. Provides travel information, trip advisories, health and safety information, climate and disaster updates, passport entry and exit requirements, for over 225 countries

p. John Hopkins University Covid Data
i. Provided detailed global and country specific covid-19 data for our heatmaps.

Challenges

Classifying articles/Data processing

Extracting the metadata from articles is an important task as we believe having a single field to identify the location and disease mentioned in the article is critical for fast database queries. Performing this task has been quite challenging as the models used are not very good identifying disease names as diseases may be abbreviated (e.g. H9N2) or may have different forms (e.g. Coronavirus vs Covid-19).

We have had more success with identifying locations as the models are more familiar with geographical names but there are also difficulties involved. Locations that are identified can be cities, states, countries or even continents which creates inconsistencies in the type of value that the field may hold. This could result in queries missing certain articles that could be relevant to the search.

Another issue related to processing the raw data from the site, is that the body of the article may mention more than 1 location or disease. Adding these values to the metadata of an article would slow down the database query as the database would need to carry out multiple comparison operations in the event of a query. The complexity of this kind of search would be O(**m** x **n**) where **m** is the average number of tags per article and **n** is the number of articles. In comparison, if an article only had 1 value per article the complexity would be O(n). In the end we settled for only having 1 report per article to maintain the speed of our search.

High Coupling

The API is over reliant on the functioning of the scraper and its consequent text analyser to perform its role accurately in order to function. Furthermore, the flask server is reliant on the correct functioning of our database so it can accurately return the correct information to the user. Evidently, each component of our system is reliant on another and a malfunction of a

single component may have a domino effect on the rest of the system and will affect the endpoint.

This problem is difficult to address due to the nature of the system. Since, our system only consists of 3-4 components (web server, database, natural language processor), we believe this is an acceptable design flaw since we have the necessary manpower and time to ensure that each component performs as intended. This problem may become more pronounced if our system is scaled up to include more components in the future.

Deployment

When we deployed our API to the web, Heroku needed a requirements.txt file which was a list of dependencies(python modules and their versions) so that it could download all the modules that were used by our API. However, we did not have this file because we were not using a virtual environment during development to isolate our dependencies for the project. The only solution was to run a script that created requirements.txt file with all Python modules that one of our team members had ever downloaded. As a result, there were many modules that our API did not use and was not required for Heroku to download. We had to manually filter all the dependencies that were relevant to our project. We have learnt from this experience and will definitely use a virtual environment to keep track of our dependencies for Phase 2.

Development of API

There were several problems we faced while creating the web server. Our biggest hurdle was the implementation of our code in general in which our spec required us to create an endpoint where users can search up articles through various fields such as date of article, location of article, disease etc.. We as a group also decided to allow users to leave some fields blank when searching up articles. This proved quite difficult to code, leading to initial difficulties during development and subsequent poorly written code. Following our first iteration, we encountered several issues in the code which made implementing extra features difficult. However in following iterations a lot of the code was cleaned up and refactored and many of the difficulties in implementing new features are no longer present.

Rescraping and formatting data

As mentioned in the Web Application Module section, rescraping of more recent and time dense (no significant gaps between dates of articles) data was needed for our disease chart to ensure the relevancy of the information it displayed. Transforming this scraped data into a format suitable for the heatmap was also required. During the scraping process it was noticed, that many articles from the second half of 2020 were missing. We theorised it was due to the depth first search nature of the webcrawling. As a result, we had to scrape the articles twice, one for scraping the 2020 articles and the other for scraping 2021 articles.

We noticed that the scraper would never progress to articles that were written after the date of the initial article so we set the starting article for the 2020 scraper to be in late december.

We made sure there were no duplicate articles by removing all 2020 articles from the scraper that scraped 2021 articles and also removed 2021 articles from the 2020 scraper. We combined the articles from these 2 scrapers to create a json file that was made up of articles starting from January 2020 with minimal gaps in between.

ISO country code mapping

One inconvenience that we encountered when integrating the disease heatmap with the disease report functionality was that the heatmap used ISO 2 letter country codes to identify countries whereas our tagged article data would use the common names for countries (e.g. China). There was a git repo which had a country code to common name mapping so we just reversed this mapping to get our desired common name to country code dictionary. https://gist.github.com/maephisto/9228207

Shortcomings

Search algorithm

Another shortcoming to our webserver is the inability to search up terms regardless of case. This was mostly due to the limitations of the mongodb collection functions we used. We discussed the alternatives of using java list functions or simply iteration via a loop through our mongodb database instead however this solution would have drastically increased the time of our queries as we would have to loop through the entire database several times when a query is made. We also discussed using certain workarounds found using regex however we quickly realised that this was very dangerous as it seemed as if it was potentially possible for our regex solution to pick up terms and articles that we did not actually want to pick up. Hence, we settled that our user must type in the exact parameter they want to match, matching our database's cases exactly.

In order to match dates however, we were forced to use the described looping solution as our scraper stores the date of an article in the database as a string. As a result, when we were to match articles to dates we had to iterate through each article in the database, manually convert them to a datetime format to then compare them to the datetimes the user has given to us. One fear of this method was the drastically increased runtime of the query. However as this looping method was only used in one function, our queries were not noticeably long.

Classification of articles

Since our Named Entity Recognition methods were not able to correctly identify the location or disease for every article in our database, a significant portion of our database lacks either both tags or one of the tags (around 3500). Therefore, we have many articles that will not get picked up in a query and are simply taking up space in our database. However, this is not without reason as every article has a **region** tag which was scraped from the article metadata. The region tag was very inconsistent in the type of values it would have. For example, it's value could be a continent (Europe), or a country (Australia) or a subcontinent (Indian subcontinent) or something random like (Medical History). However, it would provide a baseline parameter that every article would have and which a user could use to search very broadly.

We were also not able to obtain more than one reported outbreak per article as we believed that it would be more efficient performance-wise to only have 1 reported outbreak per article (as discussed in the Challenges section).

Updating the database

Due to time constraints on our project, we were not able to implement automatic scraping, tagging and importing of articles into our database so that our API could be as up to date as the website http://outbreaknewstoday.com/. We had an idea of how we could write scripts to automatically do the data processing and importing but did not know how to schedule this to happen everyday through Heroku. We decided on finishing the primary functionality of our API and writing documentation instead of developing this feature.

Difficulty loading all COVID data

The country data that was used for the Covid-19 heatmap was extremely large due to a requirement of the design of the amchart heatmap. The heatmap required the data to have the number of confirmed/recovered/deaths for each country for each date, even if the number of confirmed/recovered/deaths were all zero. This made the data have many elements that were pointless to have as they contained 0 confirmed/recovered/deaths. On top of this the earliest date in the data was around January of 2020 and the data contained the data for each country since that date. This meant that there were around (~450 days * 200 countries in the world) 90,000 objects of this format

```
{"confirmed":2077,"deaths":605,"recovered":1382,"id":"YE
"}
```

in the dataset file. As a result, the data file was around 5mb and could not be loaded during runtime. We had to cut down this file to only include the last 100 days of data so that it was small enough to be loaded during runtime.This smaller file was around 1.7mb. We had tried many other strategies including using a json file to store the data and storing the data in the original source code, but none of these methods worked.

Static Data

Our data for the heatmaps is static and stored in json/js files. Ideally, we would have an automated script which could fetch the covid data from the John Hopkins University git repo or perhaps their API so that our application always displays the most recent data. A caveat to this would be that the data needs to cut down to the last 100 days because as mentioned above, it is not possible to load the entire dataset file.

API rate limiting

Our TuGo travelsafe API limits API calls to 2 calls a second and as embarrassingly demonstrated during our final presentation, dragging the slider too quickly could cause the website to not update the travel advisory section since dragging the slider changes the date state variable which in turn triggers an API call.There is not much we can do about this, unless we contact the original developers of the API to allow us a higher a rate limit.

Summary of Major Achievements

We have had several major achievements throughout our project.

Back End Achievements

- Creation of a highly efficient automated recursive web scraper using. Python module Scrapy
- Implementation of the MongoDB database, a remote database that allows for fast parsing of our scraper results, resulting in lower load times when fetching data from our backend API
- The coding of our flask server which could handle GET requests with multiple parameters in an efficient manner, allowing for highly customised and varied requests from our endpoints.

Front End Achievements

- Sleek aesthetics whilst not sacrificing any functionality, achieved through intelligent design and UI considerations.
- Interactive heatmaps which are integrated with the the TuGo API and our in-house API.
- The usage and implementation of 3D interactive models through the usage of various libraries such as drei, react-three-fibre and gltfjsx

Improvements based on feedback

During development of the backend we have received feedback on possible implementations that we could utilise to improve our backend performance. This included the usage of pagination of our results so that users could have added customizability in the amount of results that they could see. We also received feedback on status codes to help identify the cause of problems when the back end does not work as planned, allowing for an easier time bug fixing in future development.

A key piece of feedback that we received was that due to the spreading out of features across many webpages the application felt disjointed and as a result the features felt noncohesive to a singular purpose. We were also told that our website was not focusing on the main purpose of providing travel advice. This was quickly solved in the last iteration as we incorporated most of all the features onto a singular webpage. We also made our application more intuitive by incorporating responsive web design principles such as the highlighting of text when hovered over to indicate interactivity.

Another piece of feedback we received was that our graphs may have looked intimidating to use and users may not be able to figure out how to use our heatmaps. As a result, we implemented some pop ups and had written directives on the web page to educate the user on interacting with our website.

After our final demo, we were told that it would be useful to allow the user to search up outbreaks by country on the disease heatmap because they may be geographically challenged or the country they may be looking for is small. We realised that this feature could have easily been implemented by bringing our search feature onto the disease heatmap page.

Team organisation

In allocating the work it was decided that it was most efficient to split the four group members into two groups of two and to work through any tasks in pairs with the exception of the web scraper which was developed solely by Flynn Zhang. The leader of the group was Flynn who participated in all aspects of the project and directed the workflow of the group.

The groups were structured as follows:

Group1:

- Frank Su
- Edwin Tang

Tasks allocated to this group were primarily focused on back-end development and included:

- Development of Flask server and appropriate endpoints to process HTTP requests
- Setup of MongoDB database to store data
- Development of Python scripts to access data stored in database and filter the data based upon provided filters
- Generate API documentation for use on Heroku
- Preparing slides for deliverable 3
- Wrote Swagger UI documentation

Group 2:

- Josh Rozario
- Flynn Zhang

Tasks allocated to this group were primarily focused on front-end development and included:

- Setup of API hosting on Heroku
- Development of TravelGuard using HTML/CSS/React JS
- Implementation of disease and covid-19 heatmap using amcharts onto website
- Tagging article data with country and disease name using Spacy and pycountry
- Implemented search functionality
- Implemented “Learn” page with 3D models of viral, fungal and bacterial infections
- Styling our website using Material UI
- Integration of separate TuGo TravelSafe API and in-house disease report API with heatmap
- Improvement of website ux flow as per mentor feedback
- Writing the script for deliverable 3
- Demonstrating our application in deliverable 3.

This pair structure proved to be very effective as the pairs would then further delegate tasks between themselves, preventing the risk of overlapping work whilst emphasising collaboration..

All members contributed to writing the documentation (reports) for our project.

Group meetings were not held on a set schedule but could be requested by a group member if there was an issue that they would like to talk about with the rest of the group. These meetings were primarily held using a Discord voice call.

Progress updates along with any expected delays were frequently given by the group members allowing the group to establish and adjust a rough timeline of when tasks are expected to be completed.

Major achievements in our project

By using the pair structure detailed above we were able to allocate work evenly and fairly with minimal overlap. Additionally the pair structure allowed groups to work on tasks that were suitable to their skillset. This further encouraged collaboration within the pairs ensuring efficiency in the completion of tasks.

Additionally in this project it was the first time that the members in our group developed a backend and then subsequently connected it with a frontend that we also developed ourselves. The development of a complete web application from scratch with our collaborative efforts was a major achievement of our project. This is also the first time we developed an application that was deployed to the web and could be used by other developers.

In terms of the backend, our group members previously had no experience using MongoDB but through usage of online resources we were quickly able to set up and use a MongoDB database to store and access all our data. Whilst having experience with PostgreSQL databases its implementation would have taken considerably longer as we would have needed to develop a schema thereby making MongoDB the appropriate choice for this project. The usage of a Flask server in conjunction with a RESTX extension ensured a robust server with appropriate status codes and documentation. Furthermore, the algorithm developed within the flask server allowed for highly customised API queries whilst maintaining a reasonable level of efficiency allowing for the frontend a lot of flexibility when interacting with the backend.

The processing of the scraped articles is also an achievement that is noteworthy as we did not have any experience in using Natural Language Processors and originally had no idea how we would go about extracting the country and disease name of the articles.

With our front end web-application, we had several major achievements related to the data visualisation features. One such achievement is the integration of our disease heatmap with the travel advisory API and our in-house API. Allowing the user to see health advisories along with the disease report at the click of a button is a key feature of our application. Furthermore, our covid-19 heatmap has a multitude of features: seeing country specific information, toggling between active/confirmed/recovered/deaths and animation of historical number of cases. Through the usage of libraries such as react-three fibre we have also implemented an interactive 3D model to further improve upon the aesthetics of our website. All these technical feats serve to create an overall user experience that is sure to not only pleasant and intuitive but also visually impressive.

We had no experience in implementing chart packages in React and there was a learning curve that had to be overcome in using these libraries successfully.

Skills we wish we had before this workshop

- Front end development
    - Only half the group-members have experience with front-end development. As a result, our group was not working at full strength when it came to the latter half of the project. This significantly curbed our ability to deliver more functionality in our web application.
- Web app deployment
    - None of the group members had experience with web deployment and this was a part of the project that took some time to research before we could complete this part of the project. One critical problem that we ran into during deployment was that we were not keeping track of the dependencies of our API and so our deployed “slug” had many python packages that it was not using. (More detail on this in Challenges section). We suspect this is a reason for its slow load time on heroku.

What would we have done differently

We felt that our web applications could have had more of our planned features such as introducing a predictions page which would covid progression in each country by using Tensorflow or the SIR model (as suggested by our mentors). We could have used these predictions to assign “danger values” to countries, indicating the risk of catching covid and the severity of such covid based on the reports. However, we were unable to accomplish this due to time constraints and lack of frontend development skills in our group. We should have encouraged these members to learn ReactJS earlier in the term so that they would be ready to contribute to Phase 2 of our project. Unfortunately, we did not have the foresight to do this as we were too busy focusing on our Phase 1 at the time.

We were also looking into using the google news api to provide more comprehensive results for our heatmaps. Google News API would have allowed us to search through hundreds of millions of articles from 75,000 different sources using Google’s powerful keyword search engine. We were not able to achieve this due to aforementioned time constraints and skill shortage. One thing that we did not consider doing at all, was looking to use other team’s API’s as they would have scraped a different data source to us (e.g. WHO website). Other API’s would have been useful in expanding our database of articles.

Many scraped articles had to be discarded because our natural language processors would not be able to extract out a disease or country from the article. If we were to do this again, we would try to reduce the number of discarded articles by looking into more sophisticated methods of natural language processing or we could have supplemented our dataset with another dataset which would have provided this metadata by default.

In our deliverable 2 feedback, we were told that our API only handled slightly more than 50% of test cases. We would do better next time by considering more edge cases in our API.