



**UNSW**  
SYDNEY

# University of New South Wales

Software Engineering Workshop 3

SENG3011

## DESIGN DETAILS

*Thicc Peas*

**z5238611**, Hao Cheong

**z5238064**, Brandon Green

**z519527**, Fanrui Li

**z5257129**, Ryan Vu Anh Nguyen

**z5263663**, Max Emerson Owen

## Table of Contents

1.0 Design Summary .....	3
2.0 Software Architecture .....	3
2.2.2 Tech Stack: Language .....	13
2.2.3 Tech Stack: Deployment .....	14
2.3 API Documentation Website Design .....	14
2.3.1 General Design .....	14
2.3.2 Tech Stack Language .....	15
2.3.3 Tech Stack Deployment .....	16
2.4 Website Front-End .....	16
2.4.1 General Design .....	16
2.4.3 Tech Stack Deployment .....	17
2.5 Website Backend .....	17
2.5.1 General Design .....	17
2.5.2 API Used .....	17
3.0 Architecture Summary .....	19
3.1 API Architecture .....	19
3.2 Web Application .....	20
4.0 Use cases and Requirements .....	21
4.1 API Use cases .....	21
4.2 Web Application (VaccTracc) Use cases .....	25
5.0 Final Architecture .....	32
5.1 Overview .....	32
5.2 Scraper Challenges .....	33
5.3 API Challenges .....	34
5.4 Scraper Shortcomings .....	35

## 1.0 Design Summary

For this project, our team was prescribed the [CDC](#) as its data source. The main requirements of the specification were the development of an API in Stage 1 and a web-based platform that contributes to EpiWATCH in Stage 2. It explains the tech stack chosen for each different component of each stage and the design decisions agreed upon by the team, including justification as to why we chose the tech stack as below. Research regarding other possible components on the tech stacks is made present along with comparisons between them.

For stage 2, the web application which the team ultimately agreed upon is a vaccine booking system where users are given the ability to look for the appropriate vaccines and tests in their area. The plan is to reduce the time needed on the individual behalf by providing an easy system to search and book for vaccines as well as on the clinic's behalf to promote their inventory and simplify their booking system should they need them.

## 2.0 Software Architecture

The following section explains each software component that will make up the first stage of this application. Including the tech stacks, the research, and the justification to why these stacks were chosen. This comparison are made based on criteria the team believed are the most important to consider given our time frame and budget.

### 2.1 Web Scraper Design

#### 2.1.1 General Design

The web scraper is the program responsible for receiving information from the CDC and storing it in an ideal format in our chosen database. This scraping operation will occur as follows:

- The [‘US Outbreak RSS’](#) feed will be scraped every hour.
- The [main outbreaks page](#) will be scraped every hour.

The RSS feed was chosen as it exposes all the US-based outbreaks in a very readable format, to which further links to other pages where data can be scraped from.

However, as only US outbreaks are covered by the RSS feed, the web scraper will also periodically scrape the main outbreaks page, which contains international outbreaks, every hour. An hour-long period was deemed suitable as reports on the CDC are seldom published. Thus, the period is conservative enough to save on costs and prevent overloading the CDC with requests yet frequent enough to obtain report data in a reasonable timeframe. The exact time of day are not present within the CDC reports meaning that the time frame to which they are uploaded in should also not be important enough to consider

After scraping, the outbreak information will be processed by the scraper into a format suitable for the outbreaks database and stored within the database. Figure 1 displays a tentative ER diagram for the database and the details in which we are deciding to store.

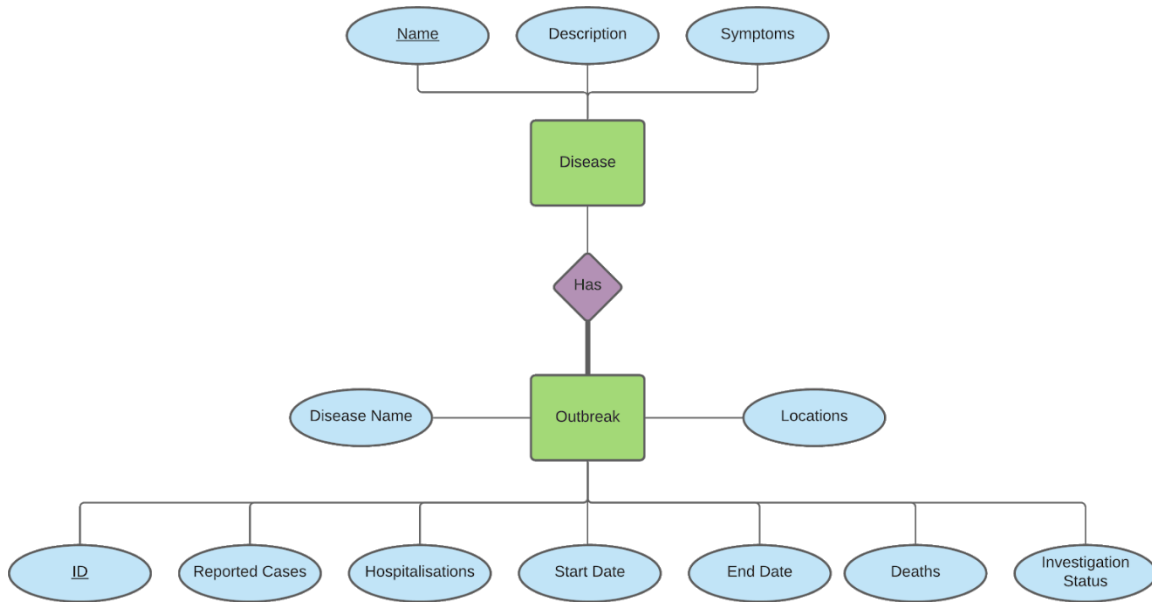
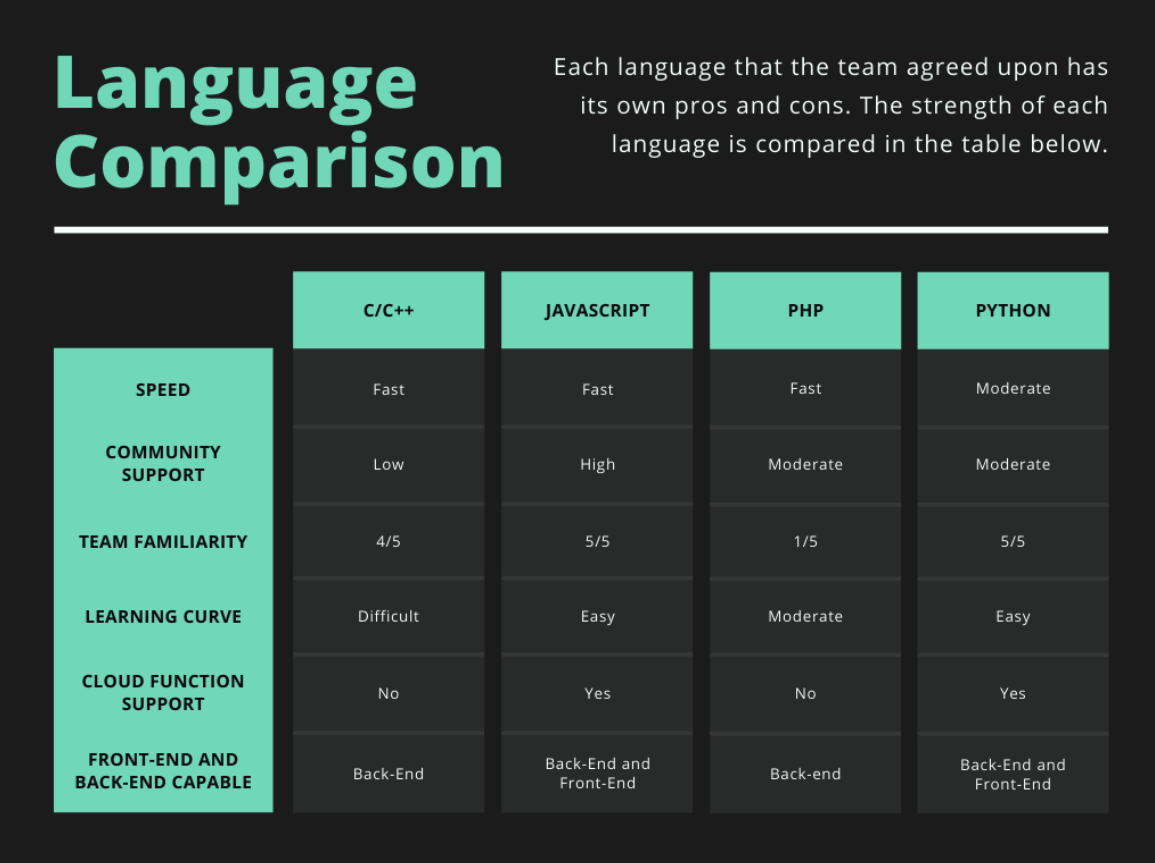


Figure 1: ER Diagram of Outbreak Database

### 2.1.2 Tech Stack: Language

In researching the development language, our team decided on a number of key factors: speed, community support, familiarity between group members, the learning curve, and cloud functionality. Ease of use as a web scraper was also considered.



Each language that the team agreed upon has its own pros and cons. The strength of each language is compared in the table below.

	C/C++	JAVASCRIPT	PHP	PYTHON
SPEED	Fast	Fast	Fast	Moderate
COMMUNITY SUPPORT	Low	High	Moderate	Moderate
TEAM FAMILIARITY	4/5	5/5	1/5	5/5
LEARNING CURVE	Difficult	Easy	Moderate	Easy
CLOUD FUNCTION SUPPORT	No	Yes	No	Yes
FRONT-END AND BACK-END CAPABLE	Back-End	Back-End and Front-End	Back-end	Back-End and Front-End

Figure 2: Comparison chart of all language choice

Ultimately, Python was chosen as it's the language the team is most familiar with and has the most experience with. JavaScript was considered due to similar familiarity, but the library prevalence and support led us to choosing Python in the end. C/C++ does not have good support for web crawler development, PHP is a language that no members on the team have experience in, and while JavaScript is best suited when dealing with basic web scraping projects, scraping large scale data from CDC is not advisable. After selecting Python as our language to program, we decided we would incorporate each of the major scraping libraries (Scrapy, BeautifulSoup4, and Selenium), utilising the strengths of each of them. The strengths and how we will use each of the libraries is detailed below:

Python Libraries	Strengths	Uses
<b>Scrapy</b>	<ul style="list-style-type: none"> <li>• Easily extensible</li> <li>• Built-in HTML and CSS data extraction tools</li> <li>• Memory and processing power efficient</li> <li>• Fast</li> <li>• Strong support for web-crawlers</li> </ul>	Will focus on crawling between the various pages of diseases.
<b>BeautifulSoup4</b>	<ul style="list-style-type: none"> <li>• Shallow learning curve, reducing the time needed for the team to learn the library</li> <li>• Easily extract data out of the requested HTML</li> <li>• Comprehensive documentation</li> <li>• Strong community support</li> </ul>	BS4 HTML parser will be useful to quickly gather all the details required of the report and given how simple it is, it will aid in development
<b>Selenium</b>	<ul style="list-style-type: none"> <li>• Automates testing for web applications</li> <li>• Beginner friendly</li> <li>• Has built in tools to mimic human interaction which allows us to have precise control over accessing the different web pages we need</li> </ul>	This will be used adjacent to Scrapy to crawl the CDC outbreak page, given its precise control over a webpage

Figure 3: Python Libraries with their strength and uses in our project

### 2.1.3 Tech Stack: Deployment

Regarding the platform used to deploy the web scraper, the team decided that serverless cloud options would be most ideal. Given that the web scraper would only run when scraping and runs periodically every hour, having a serverless architecture where the program is only executed for as long as it needs to is economical. The team had a general lack of experience with cloud computing, so familiarity with each technology wasn't a main factor. Thus, we narrowed down the choices to: Amazon Web Services's (AWS) EC2 and Lambda, and Google Cloud Platform's (GCP) Cloud Run and Cloud Function as they were the most popular and documented serverless cloud options available.

The different services which the team researched had been compared based on several criteria as seen below.  
\* Pricing values are the lowest found options

CRITERIA	CLOUD RUN	CLOUD FUNCTION	AWS LAMBDA	AWS EC2
Group Familiarity	0/6	0/6	0/6	1/6
Pricing *	Free for the first 2M Request	Free for First 2M Calls	Free for the first 1M Requests	Free for the 750 hours per month
Serverless	Yes	Yes	Yes	No
Ease of Use	Moderate / Difficult	Moderate	Moderate	Low
Difficulty triggering Cloud Events	Moderate	Easy	Easy	Moderate

Figure 4: Comparison Table of Cloud Solution to host the Web Scraper

Cloud Function was the one ultimately chosen. We saw that GCP provided the most user-friendly UI, thus lowering the barrier to entry. Additionally, GCP also offered a helpful \$300 free credit for the first 90 days as well as a money cap for some of its cloud services, making the platform seem more economical in comparison to AWS, which would immediately charge you open breaching the free tier cap. Thus, our team leaned towards a choice between the two GCP products: Cloud Run and Function.

The reason Cloud Function was chosen over Cloud Run for the following reasons:

- Cloud Function is more optimised than Cloud Run in being triggered by cloud events, specifically what the web scraper requires.
- Cloud Run is more difficult to use than Cloud Function as it requires knowledge of Docker Containers.
- Cloud Functions is more light-weight and faster than Cloud Run.

The comparison table from Google Cloud Tech in Figure 5. further showed the team that the optimal use of Cloud Functions is for data processing and 'triggered by cloud event' use cases.

Serverless Use Cases	Cloud Run	Cloud Functions	App Engine
<b>Build a web app</b>			
Web app			✓
HTTP services	✓		
<b>Developing APIs</b>			
API for web & mobile backends	✓		✓
Internal APIs and services	✓		✓
<b>Data Processing</b>	✓		
<b>Automation</b>			
Workflow & orchestration	✓	✓	
Triggered by cloud events		✓	
<b>Connecting Cloud Services</b>		✓	

Figure 5: Comparison Table of Google Serverless Use Cases (Source: Google Cloud Tech, 2020)

In order to schedule the Web Scraper to run every hour, we shall use Cloud Scheduler, another service on GCP, to do this easily.



#### 2.1.4 Tech Stack: Database

In evaluating what database to use our team compared DB services hosted on GCP as it is the deployment platform we are already working with. Figure 6. illustrates this comparison:

Database	Team Familiarity	Price	Querying Capability	Ideal Use Cases
Firestore	1/5	<ul style="list-style-type: none"><li>- Mainly charges on writes/reads/deletes</li><li>- <a href="#">Free Spark Tier</a> is suitable.</li></ul>	Advanced	<ul style="list-style-type: none"><li>- Very large data sets</li><li>- Storing complex, hierarchical data</li><li>- Advanced querying, sorting, transactions</li></ul>
Firebase Realtime Database	0/5	<ul style="list-style-type: none"><li>- Charges only on bandwidth and storage but at a higher rate.</li><li>- <a href="#">Free Spark Tier</a> is suitable.</li></ul>	Basic	<ul style="list-style-type: none"><li>- Data sets that are changed often</li><li>- Synchronising data</li><li>- Basic querying</li><li>- Frequent updates</li></ul>
Google Cloud Storage	0/5	<ul style="list-style-type: none"><li>- <a href="#">Charges based on usage.</a></li><li>- Not free.</li></ul>	Advanced	<ul style="list-style-type: none"><li>- Storing file-like data (e.g. images, binaries, etc.)</li></ul>

Figure 6: Comparison Table of Google Cloud Database Solution

Given that our database will only be occasionally updated, we opted for Firestore. Based on our research it seemed easy to use, was the cheapest of all available options, and is ideal for storing lots of data that is infrequently updated. Google Cloud Storage was immediately discounted as there wasn't a free tier option, as opposed to Firestore which has a money cap that can be sent, making it the more economical choice. Further, it has an automatic scaling option, making it ideal for storing a large amount of outbreak information.

## 2.2 API Design

### 2.2.1 General Design

The API will aim to effectively organise and disseminate the outbreak information and reports available on the CDC website. It will be an interface to the database mentioned in section 2.1, the Web Scraper Design. It will allow clients to retrieve information about:

- Diseases
- Any disease outbreaks

The specific endpoints have yet to be decided.

In terms of the API's design, the team considered two approaches towards getting the data:

1. Have the API simply access the data from the database that is filled with data from the periodic web scraper.
2. Everytime the API is called the web scraper and scrapes the requested data accordingly.

Ultimately the team chose the first approach for two reasons. The first is that if the queries are tied to the scraper, in the event that there is a large number of queries, this could result in high traffic on the CDC website themselves and incur high financial costs. The second reason is that we believe that parsing API queries into database queries would be easier in terms of translating that info rather than taking the parameters of the queries and passing them into the scraper.

In terms of the API's authorisation method, we will not be using authorisation such as HTTP Basic, JWT, API Keys or OAUTH. We concluded that the time to implement these features did not confer any noticeable benefits. We posited that if this was a production API where DDoS and high traffic usage were real factors then we would, however given that this is a prototype suce features are not required.

In terms of the design of the API documentation, we will also attempt to have the API conform to the OpenAPI standard. The reason for this decision are the following benefits:

- Following an industry standard will result in a more organised and well-thought-out API.
- OpenAPI can already be easily generated by either SwaggerHub or Stoplight.io.

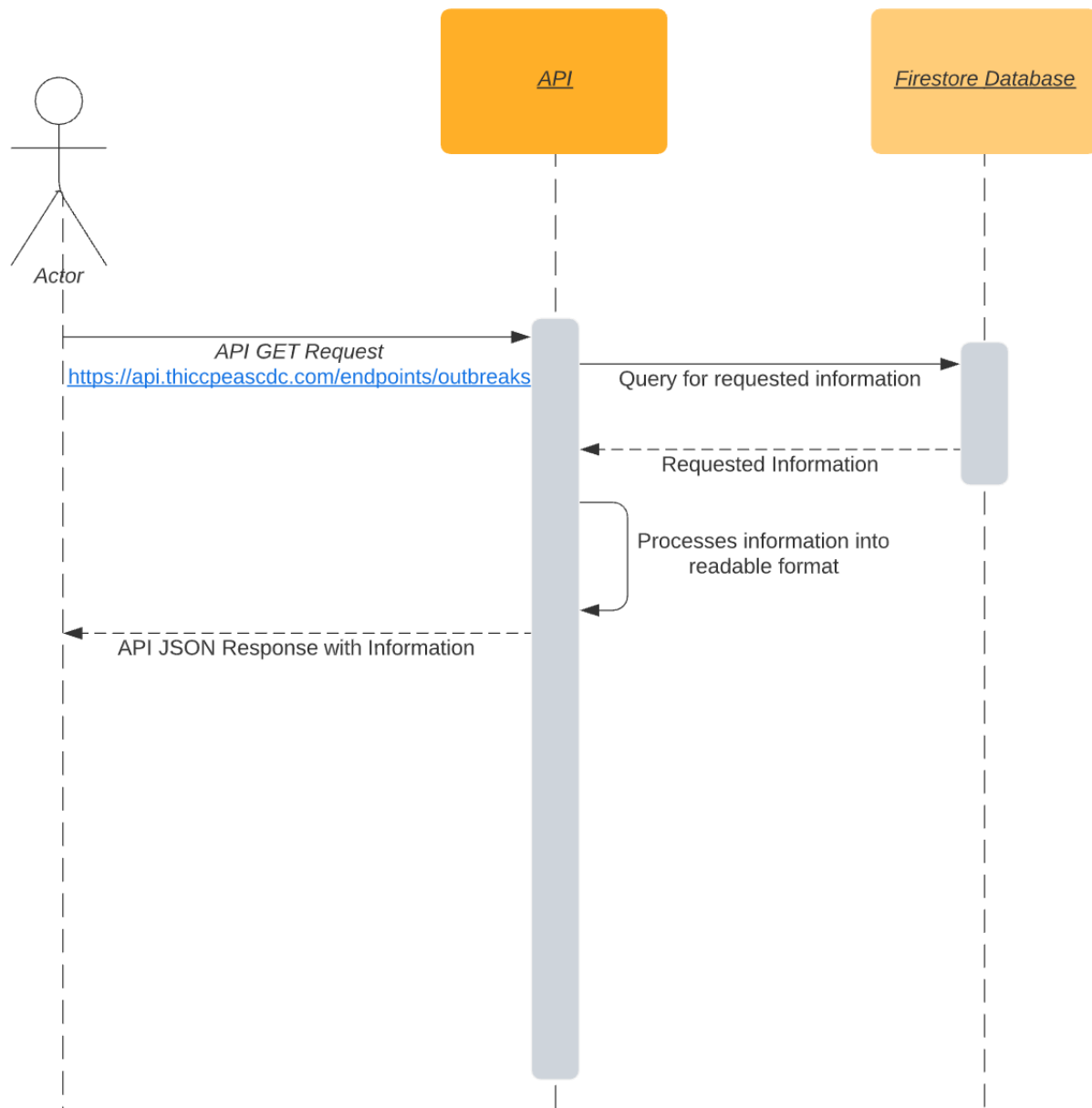


Figure 7: Sequence Diagram of an Example API GET Request

GET Request	Sample Response
<pre>GET https://www.api.thiccpescdc/endpoints/outbre content-type: application/json</pre>	<pre> 1 HTTP/1.1 200 OK 2 X-Powered-By: Express 3 Content-Type: application/json; charset=utf-8 4 Content-Length: 437 5 ETag: W/"1b5-e11Vi/mcYseZHVVP9lZR83pmuDQ" 6 Date: Fri, 05 Mar 2021 02:46:03 GMT 7 Connection: close 8 9 { 10   "outbreaks": [ 11     { 12       "id": "001", 13       "disease_name": "COVID-19", 14       "locations": [ 15         "Lidcome, NSW, Australia" 16       ], 17       "reported_cases": 17, 18       "hospitalisations": 2, 19       "deaths": 0, 20       "start_date": "07-06-20", 21       "end_date": "12-06-20", 22       "investigation_status": false 23     }, 24     { 25       "id": "002", 26       "disease_name": "Listeria", 27       "locations": [ 28         "Colorado, United States of America" 29       ], 30       "reported_cases": 2, 31       "hospitalisations": 1, 32       "deaths": 0, 33       "start_date": "07-09-20", 34       "end_date": "18-09-20", 35       "investigation_status": false 36     } 37   ] 38 }</pre>

<pre>GET https://www.api.thicpcascdc/endpoints?disease=list content-type: application/json</pre>	<pre>1 HTTP/1.1 200 OK 2 X-Powered-By: Express 3 Content-Type: application/json; charset=utf-8 4 Content-Length: 484 5 ETag: W/"1e4-Nabw/ditEKGoxM7K+NHU4IggOk" 6 Date: Fri, 05 Mar 2021 02:54:37 GMT 7 Connection: close 8 9 { 10   "name": "Listeria", 11   "description": "Listeriosis is a serious infection usually caused by eating food contaminated with erium Listeria monocytogenes. An estimated 1,600 people get listeriosis each year, and about 260 die. ection is most likely to sicken pregnant women and their newborns, adults aged 65 or older, and peopl eakened immune systems.", 12   "symptoms": [ 13     { 14       "name": "Fever" 15     }, 16     { 17       "name": "Diarrhea" 18     }, 19     { 20       "name": "Muscle Aches" 21     }, 22     { 23       "name": "Fatigue" 24     }, 25     { 26       "name": "Miscarriage" 27     } 28   ] 29 }</pre>
--	--

Figure 8: Table of Sample HTTP GET Requests with API

## 2.2.2 Tech Stack: Language

Language	Team Familiarity	Ease of Use	Amount of Online Documentation
ExpressJS/NodeJS	1/5	Easy	Substantial
ASP.NET Core	2/5	Difficult	Substantial
Flask	3/5	Easy	Substantial
PHP	0/5	Difficult	Substantial

Figure 9: Comparison Table of Possible API Languages

ExpressJS was chosen as the API framework due to its minimalistic nature ensuring ease of development. Combined with its ease of use and our team's familiarity with JavaScript in general, we decided it was the best possible option. Naturally as a result of this, NodeJS will also be used as ExpressJS requires it. Considering NodeJS' popular industry usage and extensive documentation as the back-end environment for APIs, the team deemed it beneficial in the long-term to learn and use it.

In comparison with other technologies, frameworks such as ASP.NET Core and PHP proved to be too difficult to learn and unfamiliar to use. Python's Flask library, a micro web framework that our team had experience with, was considered, however ExpressJS' more popular usage for RESTful APIs and its perceived ease of use compared to Flask caused the team to choose the latter.

### 2.2.3 Tech Stack: Deployment

To decide what platform we needed to use, the team again decided on GCP, primarily as it was already decided on for the web scraper, and consistency is ideal. We yet again used figure 5 to determine which cloud service to use, as well as our own research.

Ultimately, Cloud Run was chosen over Cloud Function and App Engine. As mentioned in the figure, Cloud Run and App Engine are the more ideal choices for APIs as compared to Cloud Function, mainly because the API requires an always-active URL clients can use to access the API. Cloud Run was chosen over App Engine as although App Engine has less latency than Cloud Run, Cloud Run is only active whenever it receives a request. This means Cloud Run does indeed have a 'cold start', resulting in some lag. This is a minor tradeoff considering the prototypal nature of the project. Thus, Cloud Run is a far more economical choice than a permanently on VPS such as App Engine.

In terms of deployment details, our team will use the official [Ubuntu Docker image](#). This is because the team is very comfortable with Linux, and it is an ideal OS for deploying web services.

## 2.3 API Documentation Website Design

### 2.3.1 General Design

API documentation provides reference for developers to the different HTTP endpoints which can be called which would primarily be GET endpoints. The different server response code will be made present for each HTTP endpoint.

Description of each API query will also be made available to the developer, allowing for quick understanding of the query's purpose. The parameters required for specific API query will be made available along with query examples and query response examples to demonstrate the correct method of usage and expected outcome.

An example of a Stoplight documentation GET page is as seen below:

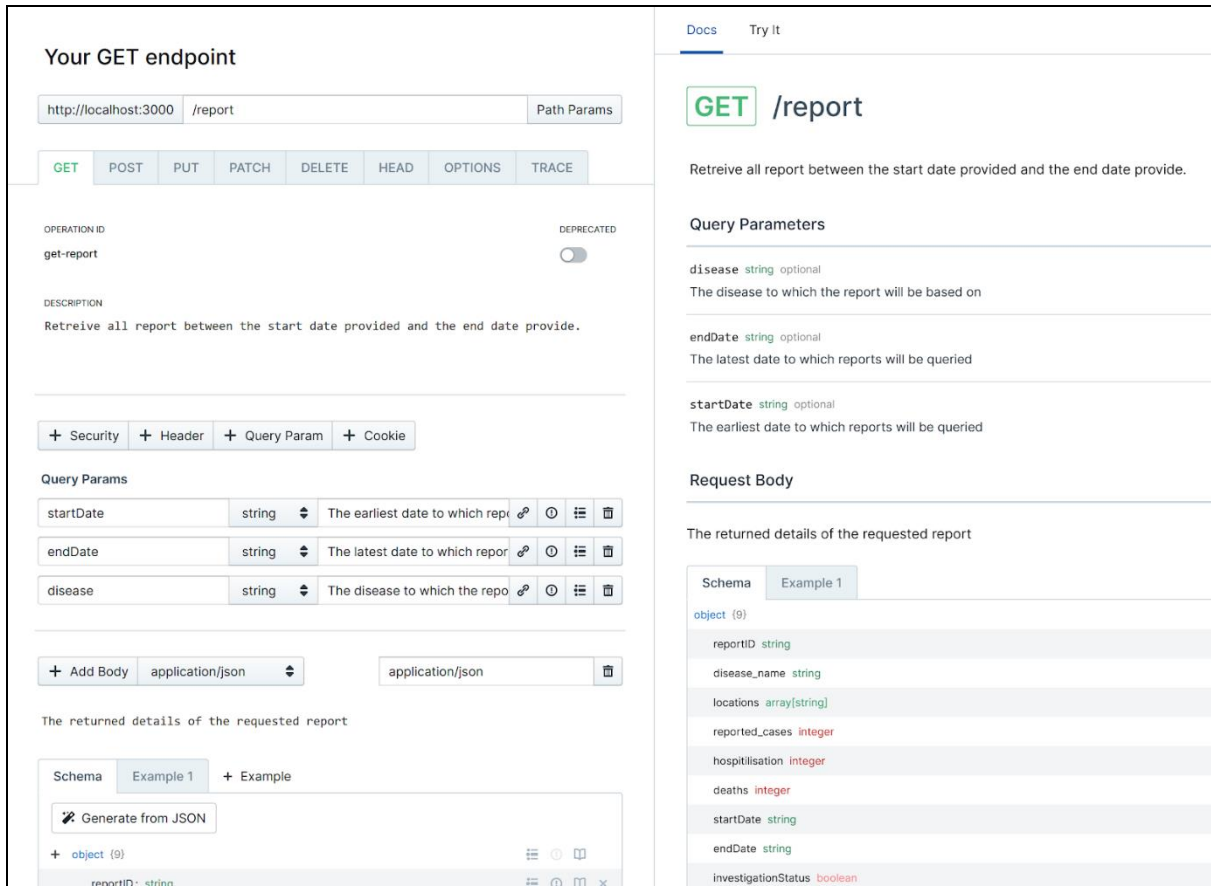


Figure 10: Example Stoplight Documentation

### 2.3.2 Tech Stack Language

In terms of the API's documentation, there were two tools for API documentation. Swagger and Stoplight. Ultimately, the team chose Stoplight as seen below:

Spotlight's primary advantage over swagger is its ability to collaborate in real time with other team members. This will speed up the writing process as multiple members can use document.

Stoplight.io	SwaggerHub
<ul style="list-style-type: none"> <li>Spotlight's primary advantage over swagger is its ability to collaborate in real time with other team members. This will speed up the writing process as multiple members can use this.</li> <li>Spotlight.io has a more easy-to-use GUI than SwaggerHub.</li> </ul>	<ul style="list-style-type: none"> <li>While collaboration is a possibility, it requires a paid subscription which is not compatible with team budget.</li> </ul>

Figure 11: Tech Stack Comparison

### 2.3.3 Tech Stack Deployment

As the API and Web Scraper is already using Cloud Run, we decided to use a platform from GCP. Using Figure 5 once again, we concluded Cloud Run would be the most economical and best choice.

Since Cloud Function simply executes functions, it is unsuitable for hosting a website. App Engine would work, however its costly 'always-on' nature ruled it out for economic reasons. Thus, the team ultimately decided Cloud Run would be the best option.

The API Documentation website will be hosted on a different Cloud Run instance than the API with a different URL. It will use a Ubuntu image for the docker container. This URL may be of the form:

<https://www.documentation.thiccpeascdc.com>.

In addition, the web server will use Express.JS to serve the web content, as it is what our team is most comfortable with, and we are already using JavaScript and ExpressJS for our API.

## 2.4 Website Front-End

### 2.4.1 General Design

The purpose design of the website has yet to be finalised, but is shall abide by the Stage 2 project specification and use APIs to accomplish its goal of contributing the UNSW's epiWATCH.

Web Framework	Familiarity	Ease of Use
React	4/5	Moderate
Vue	1/5	Moderate
ASP.NET	1/5	Difficult

*Figure 12: Comparison Table of Possible Web Frameworks*

Ultimately, React was chosen as it was the most familiar to our team members, allowing us to create a website efficiently.



### 2.4.3 Tech Stack Deployment

In order to keep the architecture simplified, our website shall use the same GCP platform as the previous services. Using Figure 5. again, we ultimately decided on Cloud Run, for reasons largely similar to the previous services. Whilst Cloud Functions is simply not sufficient for a website and App Engine being too pricy as an 'always-on' service, Cloud Run remains the most economical and efficient choice for the team, especially since the team is already using it.

In addition, the web server will use ExpressJS, as it is what our team is most comfortable with, and we are already using JavaScript and ExpressJS in our architecture.

We plan for the website to be hosted on a separate URL in a separate Cloud Run instance. The Docker container will be the official [Ubuntu Linux](#) image as the team is comfortable with Linux and Linux is ideal for hosting a website.

## 2.5 Website Backend

### 2.5.1 General Design

The website's backend, created with NodeJS and ExpressJS, acts as the link between our team's CDC API and the Google Maps API for important functionality in the front-end:

- Vaccine Finder
- Symptom Checker

It also acts as the link between the application and the database in order to fetch clinic information for the front-end.

### 2.5.2 API Used

For our application, we chose to use two APIs: our CDC API and the Google Maps API.

The symptom checker feature applies the CDC API. User symptoms are cross-referenced with disease symptoms CDC API to query the database for the appropriate disease(s) the user may be afflicted with.

For the Google API, we use their geolocation feature to parse the addresses of the clinics and retrieve the clinic's coordinates which is later stored in the Firestore database when required by the front-end. An important note is that the geolocation of each address is calculated only upon the following conditions: a clinic being added, or a clinic being updated. This is because using the Geolocation API can be costly, and the latitude and longitude of a clinic is likely to change in real-time, so the backend simply calculates the geolocation upfront and stores it in the database, as opposed to calculating it on-demand. With this information, the database will be queried and be used to display clinics on the "Vaccine Finder" page. To be able to display the map and the coordinates in our database, we used the [Google Map React](#) NPM package to achieve this visualisation.

Here are the list of final endpoints of the backend:

Endpoint	Description
<i>GET /log</i>	Acts as a wrapper to the relevant CDC API endpoint.
<i>GET /reports</i>	Acts as a wrapper to the relevant CDC API endpoint.
<i>GET /report/:reportid</i>	Acts as a wrapper to the relevant CDC API endpoint.
<i>GET /diseases</i>	Acts as a wrapper to the relevant CDC API endpoint.
<i>GET /disease/:diseaseid</i>	Acts as a wrapper to the relevant CDC API endpoint.
<i>GET /articles</i>	Acts as a wrapper to the relevant CDC API endpoint.
<i>GET /article/:articleid</i>	Acts as a wrapper to the relevant CDC API endpoint.
<i>GET /signup</i>	Signs up a clinic given the correct details.
<i>GET /login</i>	Logs in a user given the correct details.
<i>GET /getclinics</i>	Returns all clinics from Firestore database.
<i>GET /getvaccines</i>	Returns all vaccines from Firestore database.
<i>GET /gettests</i>	Returns all tests from Firestore database.

## 3.0 Architecture Summary

### 3.1 API Architecture

In summary, our API application is mainly hosted on GCP. It will periodically scrape the CDC website for information and deposit that into the Firestore database. Upon client request, the API shall access this database and return the relevant information. For reference, *Figure 13* depicts the broad architecture of the proposed system.

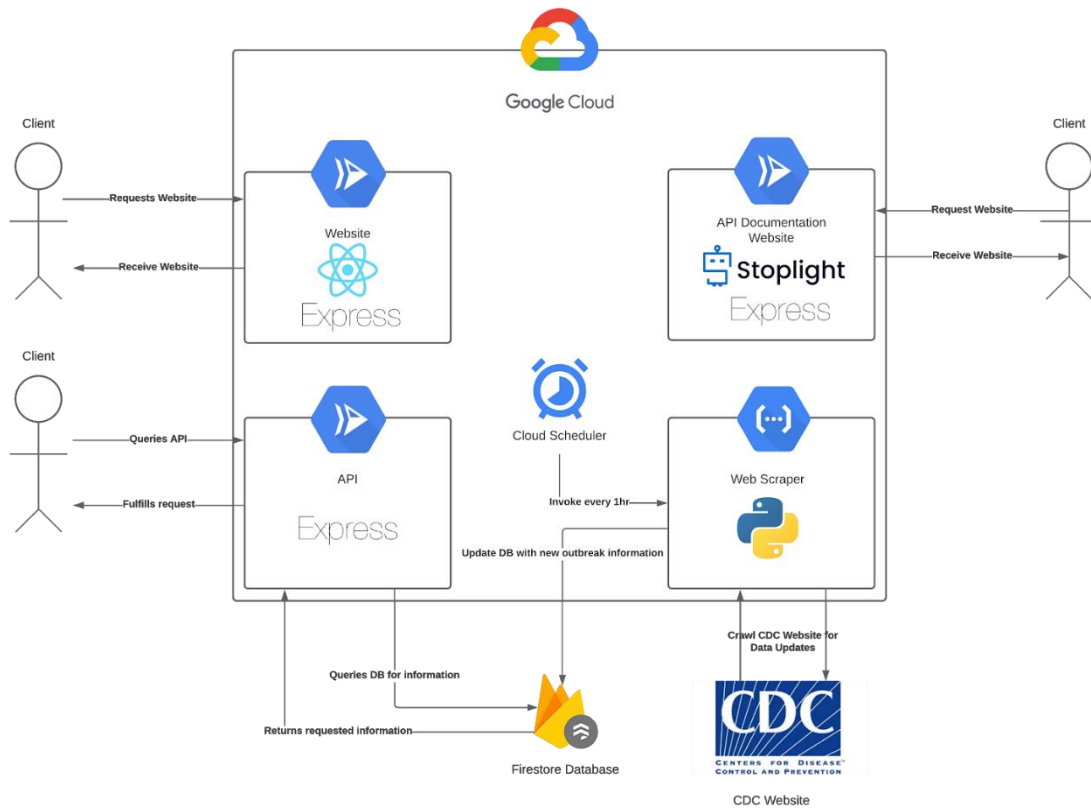
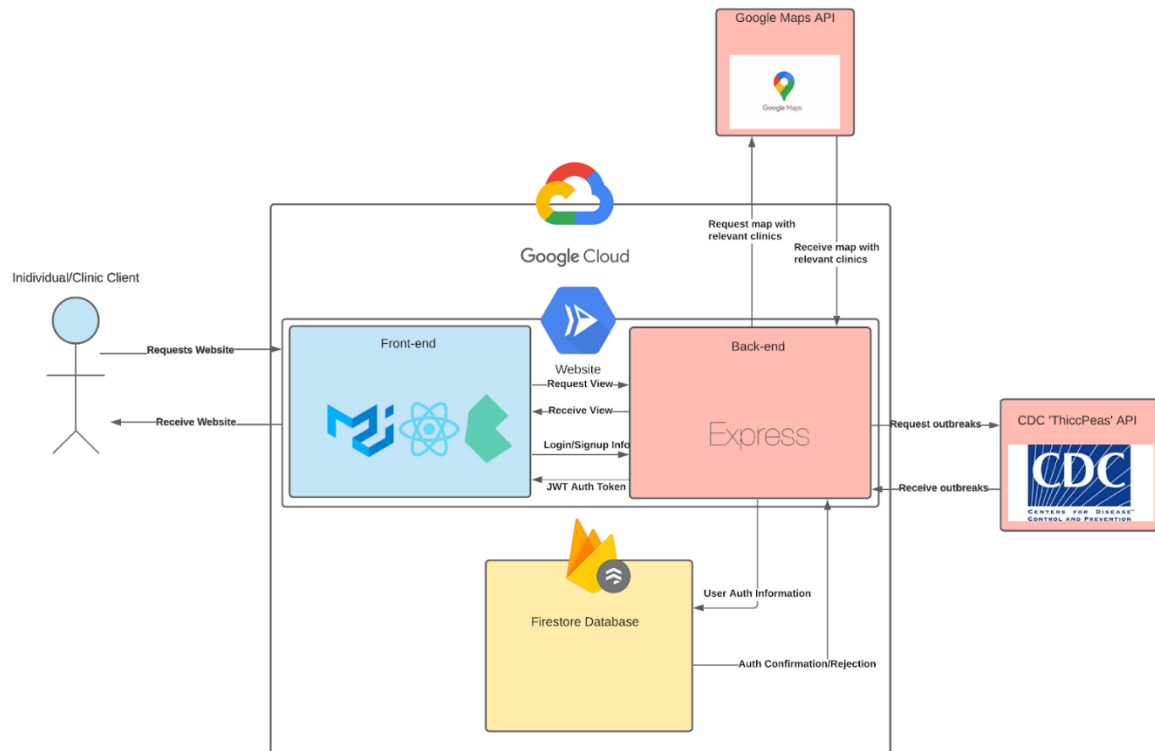


Figure 13: System Architecture Diagram

### 3.2 Web Application

Our web application is also similarly hosted on GCP with a similar tech stack. As shown below, the ExpressJS backend serves as the connector between the Google Maps and Thicc Peas APIs, as well as the link between the application and the database. The front-end is made with ReactJS, CSS, and MaterialUI. Both backend and frontend are designed to be deployed on the 'Cloud Run' platform



## 4.0 Use cases and Requirements

### 4.1 API Use cases

Below are all the use cases for the API. It includes a description of a situation where a user would want to use the system, the endpoint, and the steps necessary to run the endpoint.

<b>Name:</b> API querying for disease reports based on a given disease name
<b>Description:</b> A user wants to search for a disease report with either a specific disease name or a general disease category
<b>Endpoint:</b> <a href="https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/diseases?disease_names=#">https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/diseases?disease_names=#</a>
<b>Steps:</b> <ul style="list-style-type: none"><li>• A disease name is provided to the endpoint, where {#} is the provided disease name</li><li>• If the disease name provided is valid, disease ID, symptoms, and their reports will be returned to the user</li><li>• If the disease name is invalid, a blank JSON object is returned</li><li>• If no disease name is provided, all disease reports are returned</li></ul>

<b>Name:</b> API querying for disease reports based on a given disease ID
<b>Description:</b> A user wants to search for a disease report with a specific disease ID
<b>Endpoint:</b> <a href="https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/disease/#">https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/disease/#</a>
<b>Steps:</b> <ul style="list-style-type: none"><li>• A disease ID is provided to the endpoint, where {#} is the provided disease ID</li><li>• If the disease ID provided is valid, disease ID, symptoms, and reports of that specific disease will be returned to the user</li><li>• If the disease ID is invalid, a 404 error is returned along with an error message</li></ul>

<b>Name:</b> API querying for articles based on a given time frame
<b>Description:</b> A user wants to search for an article within a specific time frame
<b>Endpoint:</b> <a href="https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/articles?start_date=#1&amp;end_date=#2">https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/articles?start_date=#1&amp;end_date=#2</a>

**Steps:**

- The start and end dates are provided to the endpoint where {#1} and {#2} are the start and end dates respectively
- If the start and end dates are valid, the article ID, the headline, the main text, the URL of the article, and the reports related to the article are returned to the user
- If the provided dates do not have any available articles, a blank JSON object is returned to the user
- If the dates provided are invalid (end-date before start date), a 400 error is returned along with the error message
- If only the start date is provided, it will return all the articles after the start date
- If only the end date is provided, it will return all the articles up to the end date
- If no dates are provided, it will return all the articles available in our database

**Name:** API querying for articles based on the specified keyword(s)

**Description:**

A user wants to search for an article that mentions a specified key term

**Endpoint:**

[https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/articles?key\\_terms=#](https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/articles?key_terms=#)

**Steps:**

- The key terms are provided to the endpoint where {#} is the specified key terms. Multiple key terms are delimited by a comma (,)
- The key terms specified and are matched to articles with the key terms in their main text.
- If the key term(s) exists it will return
- The article ID, headline, main text, URL of the original main text, and all the reports related to the article will be returned to the user
- If the key terms(s) do not exist, a blank JSON object is returned to the user.
- If no key terms were provided, all articles would be returned to the user

**Name:** API querying for articles based on specified article ID

**Description:**

A user wants to search for an article based on a specified article ID

**Endpoint:**

<https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/article/#>

**Steps:**

- A specific article ID is provided to the endpoint where {#} is the article ID
- If the article ID is valid, the article along with the headline, main text, and URL will be returned to the user.
- If the article ID is invalid, then a 404 error will be returned along with an error message in the response body

<b>Name:</b> API querying reports based on a given time frame
<b>Description:</b> A user wants to search for reports between a given start date and end date
<b>Endpoint:</b> <a href="https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/reports?start_date=#1&amp;end_date=#2">https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/reports?start_date=#1&amp;end_date=#2</a>
<b>Steps:</b> <ul style="list-style-type: none"> <li>• The start and end dates are provided to the endpoint respectively to {#1} and {#2}</li> <li>• If the dates are valid, the reports between said given time will be returned along with the report's data and the report ID</li> <li>• If there are no reports between the start and end dates, a blank JSON object will be returned</li> <li>• If the dates are invalid, a 400 error would be returned along with the error message</li> <li>• If only the start date was provided, only reports after the start date will be returned</li> <li>• If only the end date was provided, only reports prior to the end date will be returned</li> <li>• If no dates were provided, all reports in the database will be returned</li> </ul>

<b>Name:</b> API querying reports based on a specific location
<b>Description:</b> A user wants to search for reports given a specific country
<b>Endpoint:</b> <a href="https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/reports?location=#">https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/reports?location=#</a>
<b>Steps:</b> <ul style="list-style-type: none"> <li>• The country's name is provided to the endpoint where {#} is the country which the reports to be queried are from</li> <li>• If the country is valid, the report along with the report data and the report ID will be returned</li> <li>• If the country does not exist, a blank JSON object will be returned</li> <li>• If no country is provided, then reports from any country will be returned</li> </ul>

<b>Name:</b> API querying reports based on the specified keyword(s)
<b>Description:</b> A user wants to search for reports based on one or more keywords
<b>Endpoint:</b> <a href="https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/reports?key_terms=#">https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/reports?key_terms=#</a>
<b>Steps:</b>

- The specified keyword is provided to the endpoint where the {#} is the keyword. Multiple keywords are delimited by a comma (,).
- If the specified keyword(s) are valid, reports containing the keywords in their main text will be returned to the user
- If the specified keyword(s) are not mentioned in any main text, a blank JSON object will be returned
- If no keywords were provided, then all reports regardless of their content will be returned.

**Name:** API querying reports based on a report ID

**Description:**

A user wants to search for a report with a specific report ID

**Endpoint:**

<https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/report/#>

**Steps:**

- The specified report ID is provided to the endpoint where {#} is the report ID
- If the report ID is valid, the specified report will be returned
- If the report ID is invalid, a 404 error along with an error message will be returned
- If no report ID is provided, then all report regardless of their ID would be returned

**Name:** API querying the logs of recent calls

**Description:**

A user wants to view the logs of the most recent API calls

**Endpoint:**

<https://thicc-peas-cdc-api-o54gbxra3a-an.a.run.app/log>

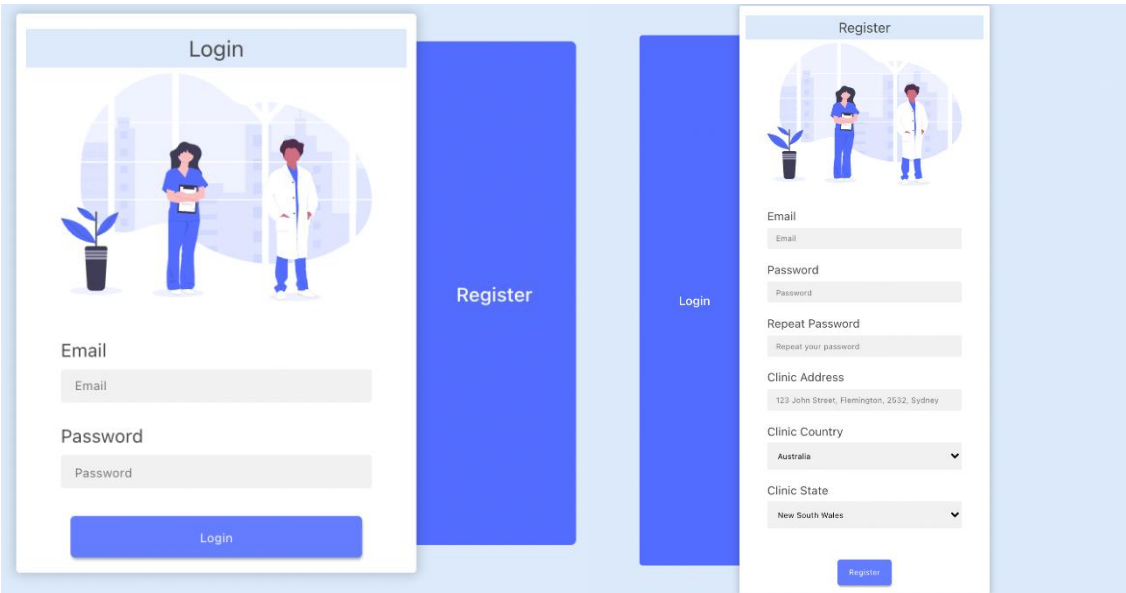
**Steps:**

- Once the endpoint is executed, all the logs of the most recent API calls would be returned in a JSON object



## 4.2 Web Application (VaccTracc) Use cases

Below are a list of use cases. It includes a description of a scenario

<b>Feature:</b> Registration with the application
<b>Description:</b> This allows medical institutions such as Clinics and Hospitals to register an account and gain access to the inventory dashboard as well as setting up an email booking system should they choose to.
<b>What problem it solves:</b> Provides the medical institutions with a simple promoting platform such that any individual who wants to book a vaccine and or test can search for institutions that only have the vaccines they want. Reduces any unnecessary time wasted by the clinics to decline individuals who want vaccines they do not have.
 <p>The image displays a mockup of the VaccTracc web application interface. It features two primary screens: a 'Login' screen on the left and a 'Register' screen on the right. The 'Login' screen includes a header with the title 'Login', an illustration of two medical professionals, and input fields for 'Email' and 'Password', followed by a 'Login' button. The 'Register' screen has a header with the title 'Register', the same illustration, and input fields for 'Email', 'Password', 'Repeat Password', 'Clinic Address', 'Clinic Country' (with a dropdown menu showing 'Australia'), and 'Clinic State' (with a dropdown menu showing 'New South Wales'), followed by a 'Register' button. A central blue vertical bar separates the two screens, with the word 'Register' on the left side and 'Login' on the right side.</p>

<b>Feature:</b> Inventory Dashboard
<b>Description:</b> This allows medical institutions to decide what vaccines and tests they want to promote as well as their current inventory.
<b>What problem it solves:</b>

Provides the medical institutions with the ability to both keep track of their inventory as well as decide what vaccines and tests they would want to promote to an individual. Also provides an inventory management system to medical institutions that do not have one.

## Clinic Dashboard

Inventory			
Disease	Type	Amount	
Covid	Test	20	<input type="checkbox"/>
Covid	Vaccine	50	<input type="checkbox"/>
Covid	Vaccine	50	<input type="checkbox"/>

**Add to Inventory**

**Disease**

**Type**

**Amount**

Add

0 inventory selected

### Feature: Booking System (by Email)

#### Description:

Clinics can, after signing up, keep track of clients who have booked through the application system on the dashboard can keep track of all their information

#### What problem it solves:

This provides a booking system to clinics, especially, newer practices, with a pre-made booking system for those who want them, reducing the unnecessary cost of building a bespoke booking system.

Home
Bookings
Inventory
Logout

## Clinic Dashboard

Recent Bookings					
No.	Name	Phone	DOB	Type	Disease
1	Giuseppe Macklemore	0421 519 231	8/12/1993	Vaccine	Covid
2	Sarah Jones	0462 953 172	2/2/2002	Vaccine	Covid
3	Chun Li	0451 825 231	1/3/1968	Vaccine	Covid
4	Johney Appleseed	0462 981 682	7/6/1987	Vaccine	Covid
5	Barackus Pngbloom	0481 421 862	12/3/2001	Vaccine	Covid

**Add to Booking**

**Name**

**DOB**

**Phone**

**Disease**

**Type**

Add

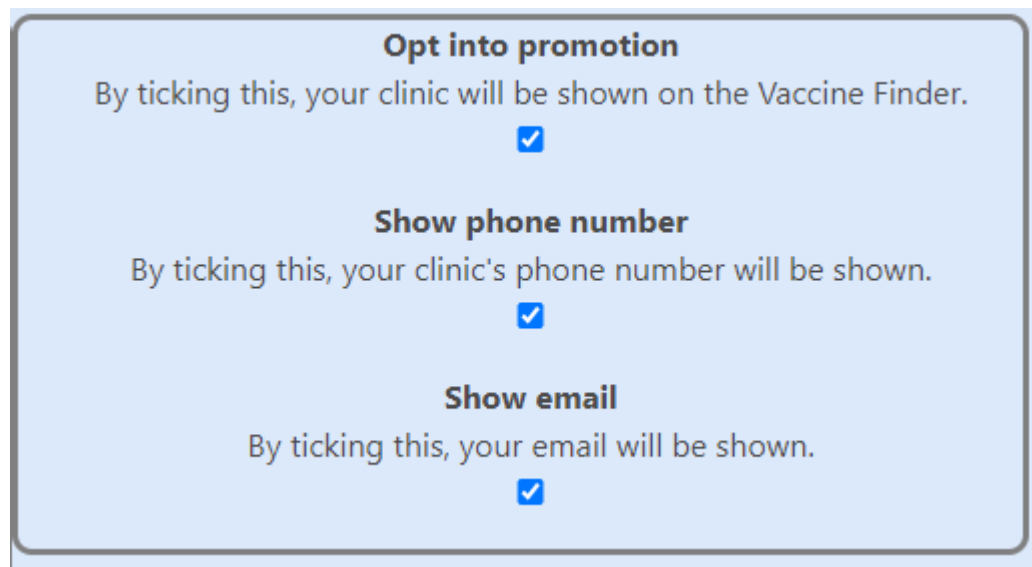
0 bookings selected

**Feature:** Booking System (by Phone)**Description:**

Clinics who decide to opt-out of our booking system can decide to have their phone number be displayed instead to clients.

**What problem it solves:**

This provides the clinics with the advantages of still using our promoting platform without the need for them to learn and train with a new system given that they have their own working booking system. Also promotes their contact detail in an easily accessible location.



The screenshot shows a light blue rounded rectangular box containing three sections, each with a title, a description, and a checked checkbox:

- Opt into promotion**  
By ticking this, your clinic will be shown on the Vaccine Finder.  
☒
- Show phone number**  
By ticking this, your clinic's phone number will be shown.  
☒
- Show email**  
By ticking this, your email will be shown.  
☒

**Feature:** Check Availability map page**Description:**

Individuals can enter all the tests and or vaccines that they want and their options will be queries against the database of clinics with those tests and vaccines. Clinics with stock will be displayed on a map. Individuals can then through the page, select the clinic which is the best suited to them and book accordingly.

**What problem it solves:**

Provides an easy way for individuals to search for clinics and vaccines in their area without the need to do research and phone individual clinics just to be put on hold or be told that the clinic no longer has a stock of what they need. Reduce the time wasted by the individual which could be better used to get the necessary vaccines or tests.

VACCINE FINDER  
SYMPTOM CHECKER

☐ Select All  
☒ COVID-19  
☐ Measles  
☐ Flu  
☐ Hepatitis B  
☐ Rotavirus

Find Available Clinics

VACCINE FINDER  
SYMPTOM CHECKER

Find Available Clinics

**Lidcombe Family Medical Centre**  
Shop 38/92 Parramatta Rd, Lidcombe NSW 2141  
**Contact Info**  
Ph: (02) 8022 8442  
Email: lidcomefamilyhealthcentre@outlook.com  
**Opening/Closing Times**  
Sunday: Closed  
Monday: 9am-5pm  
Tuesday: 9am-5pm  
Wednesday: 9am-5pm  
Thursday: 9am-5pm  
Friday: 9am-5pm  
Saturday: 9am-3pm

**Redfern Community Health Centre**  
103-105 Redfern St, Redfern NSW 2016  
**Contact Info**  
Ph: (02) 9395 0444  
Email: redfermhealthcentre@outlook.com  
**Opening/Closing Times**  
Sunday: Closed  
Monday: 8:30am-3pm  
Tuesday: 8:30am-3pm  
Wednesday: 8:30am-3pm  
Thursday: 8:30am-3pm  
Friday: 8:30am-3pm  
Saturday: Closed

**Royal North Shore Hospital**  
Reserve Rd, St Leonards NSW 2065  
**Contact Info**  
Ph: (02) 9926 7111  
Email: NSLHD-Chatback@health.nsw.gov.au  
**Opening/Closing Times**  
Sunday: 24 hrs  
Monday: 24 hrs  
Tuesday: 24 hrs  
Wednesday: 24 hrs  
Thursday: 24 hrs

## Feature: Symptom Checker

### Description:

Individuals who are experiencing symptoms of a disease but who might not be informed of various diseases and their symptoms can visit the symptom checker page. The page should display

diseases from CDC API and allow users to select symptoms they have. If a user discovers they have a majority of the symptoms, they are recommended to use the vaccine finder to find a clinic with a suitable test or clinic, or seek other medical guidance.

**What problem it solves:**

Provides a simple way to inform the individuals who are unfamiliar with common diseases with their symptoms and allowing them, through our system, to make a more informative booking about what potential test they might need.

**Feature: Individual Booking (Email)**

**Description:**

Users can select a clinic that uses our application's email booking system. User will enter their details such as their names, date of birth, email, phone number, date of appointment and send a booking request which our system will formulate into an email.

The clinic will review the client's details and confirm or decline the appointment request accordingly. The user will then receive an email either informing about the success or decline of the booking along with a cancellation link to cancel their booking. A day prior to their appointment to appointment, the user will be sent a reminder email along with another cancellation link.

**What problem it solves:**

Provides the individual with a quick and simple method of booking through our application, reducing the time needed to find the appropriate contact details, physically calling the clinic, and waiting for them to confirm the appointment time. Also provides a reminding system to individuals who may need a reminder of their appointment to prevent missing them.

**Feature:** Individual Booking (Phone)

**Description:**

Individuals can select clinics that have chosen not to adopt the application booking system and instead use their traditional phone-in method. After finding a suitable clinic, the phone number of the said clinic will be displayed to the user who can then call and make the booking

**What problem it solves:**

Provides individuals who do not have an email account to have still have a method of booking traditionally if they want to. It also provides an easy way for the client to find the necessary contact details to make a booking.

Phone Booking  
Lidcome Family Medical Centre



Phone Number

(02) 8022 8442

Email

## 5.0 Final Architecture

### 5.1 Overview

Our software architecture has not changed significantly from our planning stage. We did decide to use React.JS instead of just Javascript, but other than that, our plan has been implemented

How we formatted the

The team since then has implemented an API including all the relevant scrapers necessary. Below are the different challenges that were faced during the development as well the various shortcomings in regards to what was planned and what we ultimately had.

The ER diagram was redesigned:

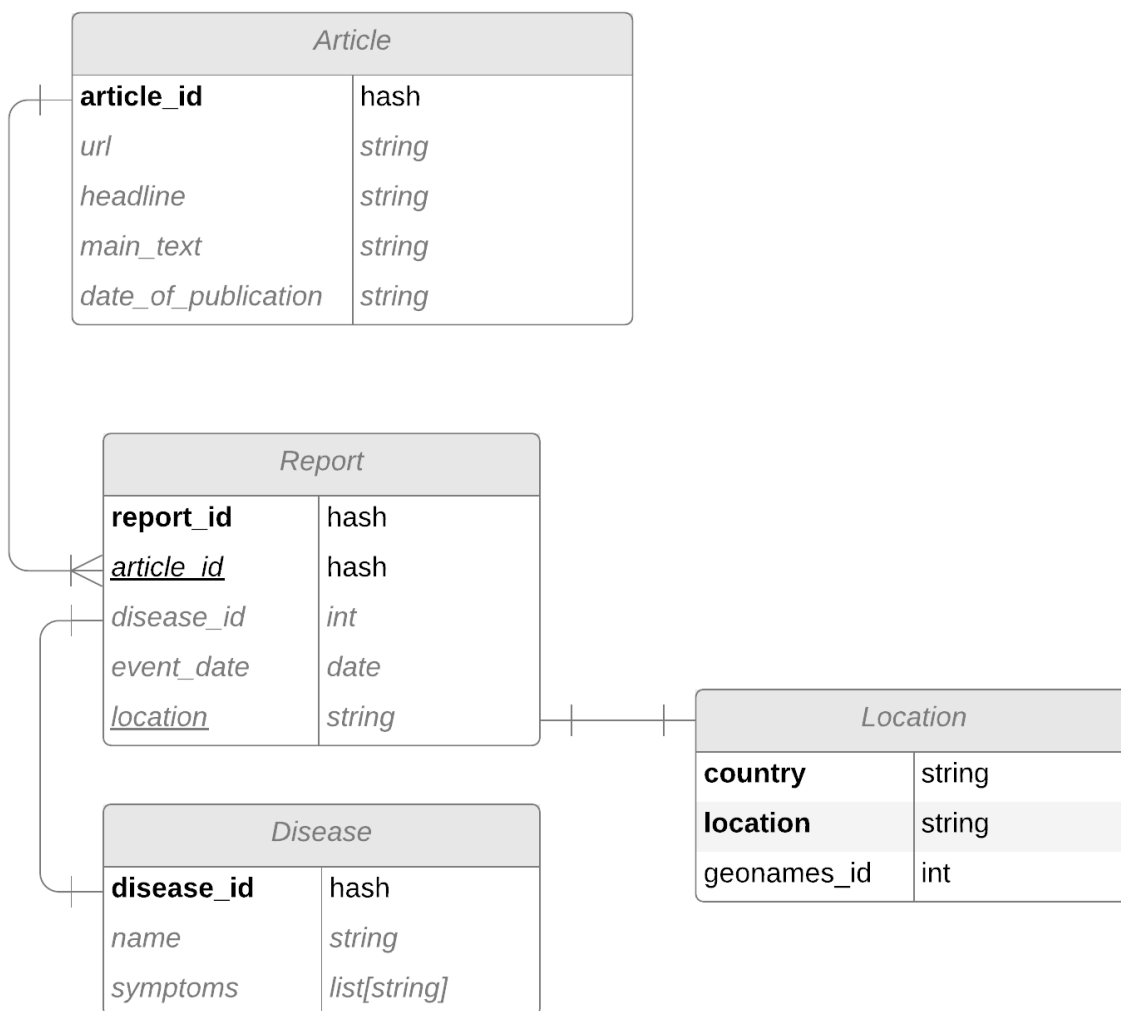


Figure 14. Final Diagram



One issue that the team came across was the updating of the database. Given how the scrapers are written, the scrapers will scrape every report they could every time. This meant that on a consecutive scrape the majority will contain duplicates.

One suggestion was to rebuild the database each time the scrape ran but as time goes on, the database will progressively get bigger and uploading of the data will get progressively longer. Instead, the team decided upon using hashing to help with our duplicates. Every time the scraper is run, the content of each article is concatenated and hashed using the MD5 hash and the resulting hash value acts as the id of the article. During upload, the hash of each article is compared with those in the firestore and if they already exist, the article will not be uploaded. The choice of an MD5 hash is its variable to fixed-length output and its ease of use.

## 5.2 Scraper Challenges

Given the data source that we were given, the web scrapers that we have running required a lot of workarounds to get workings. The biggest issue was the inconsistency of the CDC pages regarding outbreaks.

The first inconsistency of the page locations. Since many pages do not share a consistent path to their resource, we had to manually find all possible structures of the URL and then account for their differences. Certain outliers of reports pathways were unscrapable given how inconsistent they were but we managed to get the large majority of scrapable pages

Secondly, some diseases did not actually have any outbreak reports. So the team chose based on the list given to us in the specification to search for all diseases which did report outbreaks and wrote scrapers for those diseases. Disease such as histoplasmosis does not have any outbreak pages to scrape from so it is not scraped

Thirdly, since the wording and the HTML structure of the web pages are different, certain specific pages required the scraper to scrape different class types to get the required information. A massive challenge was to make sure we accounted for as many as we can to ensure the database from which the API is calling is as complete as can be.

```
#Returns True if the page uses "Reported Cases", else it uses "Case Count"
def RcChecker(soupHandler):
    aag=soupHandler.findAll('div', class_="card-body bg-tertiary")
    reportedCaseFilter = re.compile(".*Reported Cases.*")
    lastDiv = aag[1:-1]
    if lastDiv==[]:
        lastDiv = aag[1:]
    for i in lastDiv:
        liTags = i.find_all("li")
        if (liTags!=[]):
            for li in liTags:
                if (reportedCaseFilter.match(li.text.rstrip().lstrip())):
                    return True
            return False

#Return the main report content of the page (I.E As of March 19, 2020 there has been a large influx of...)
def getMainText(soupHandler):
    aag=soupHandler.find_all('div', class_="card-body bg-white")
```

Figure 15: Difference in pages given their "At A Glance" Box

### 5.3 API Challenges

The firestore database which we used could not run complicated queries like those of SQL-based languages. This caused difficulties when attempting to retrieve results from the database to send to the user. Problems include the inability to perform inequality checks for multiple fields, substring checking, and regex operation. For example, here we show that when trying to query with 'start\_date', 'end\_date' and 'location', Firestore will not be able to execute the compound query as it queries multiple fields with inequalities.:

```
...// Get reports according to the correct date ranges.
...let reportQueryRef = db.collection(FS_REPORTS_COLLECTION);
...if (start_date) reportQueryRef = reportQueryRef.where('event_date', '>=', start_date);
...if (end_date) reportQueryRef = reportQueryRef.where('event_date', '<=', end_date);
...if (location) reportQueryRef = reportQueryRef.where('location', '>=', location).where('location', '<=', location + '\uf8ff');
```

To fix this, the majority of the checks have to be done on the client-side to be able to account for those complexities as seen below:

```
...// Get reports according to the correct date ranges.
...let reportQueryRef = db.collection(FS_REPORTS_COLLECTION);
...if (start_date) reportQueryRef = reportQueryRef.where('event_date', '>=', start_date);
...if (end_date) reportQueryRef = reportQueryRef.where('event_date', '<=', end_date);

...reportQueryRef.get().then(queryResults => {
...  let report_result_promises = [];
...  let report_results = [];

...  queryResults.forEach(doc => {
...    // Create a promise for every report that is going to be added.
...    report_result_promises.push(new Promise((resolve, reject) => {
...      reportData = doc.data();

...      // Create a report_result, which is a combination of a report
...      // with its disease and article. This is what we shall
...      // send back to the user.
...      create_report_result(reportData).then(reportResult => {
...        reportData = doc.data(); // Have to recalculate this ofr some reason. If not then the value is cached across all docs.
...        articleData = reportResult.article;
...        diseaseData = reportResult.disease;

...        // Check location query parameter.
...        if (location) {
...          if (!reportData.location.toLowerCase().includes(location)) {
...            resolve();
...            return;
...          }
...        }
...      })
...    })
...  })
...})
```

Here, we had to check the 'location' query parameter once we got all the results for the Firestore query. This poses some small performance problems, however, there is no real workaround this issue with Firestore.

## 5.4 Scraper Shortcomings

CDC scraper had a multitude of issues which resulted in our inability to properly scrape everything to a degree which we were satisfied with. An example of this was for trying to scrape older records of certain diseases, the reference pages containing scanned versions of the reports making them unscrapable. This resulted in only the reports up to a certain year beginning to become scrapable, meaning those reports that came before are not accounted for in the database.

Another issue which we faced was certain diseases themselves did not actually have any outbreak reports as they are either too common to have easily trackable reports or the cases occurring are so few and far between that the reports were aggregated into one data point. Certain outbreak reports were also too vague to scrape the required info, such as titles or main text. There was also no consistent way to scrape all the outbreak reports given that their titles are all worded very differently to find a coherent way to scrape them.

Ultimately, the nature of the CDC website meant that the team was unable to create a master scraper that is able to scrape all the details needed for our API. Instead, the team opted to create individual scrapers for scrapable diseases available on the CDC which is then hosted in Google Cloud Function which will run on a weekly bases

## 5.5 API Shortcomings

Our API does not currently have any security measures in place. While our database can't be influenced from the API, as we only have GET requests, there is nothing stopping a malicious user from overloading our API with bogus requests.