University of New South Wales
Software Engineering Workshop 3
SENG3011

Design Document

Thicc Peas
z5238611 Hao Cheong
z5258064 Brandon Green
z5191527 Fanrui Li
z5257129 Ryan Vu Anh Nguyen
z5263663 Max Owen

Table of Contents

1.0 Design Summary

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

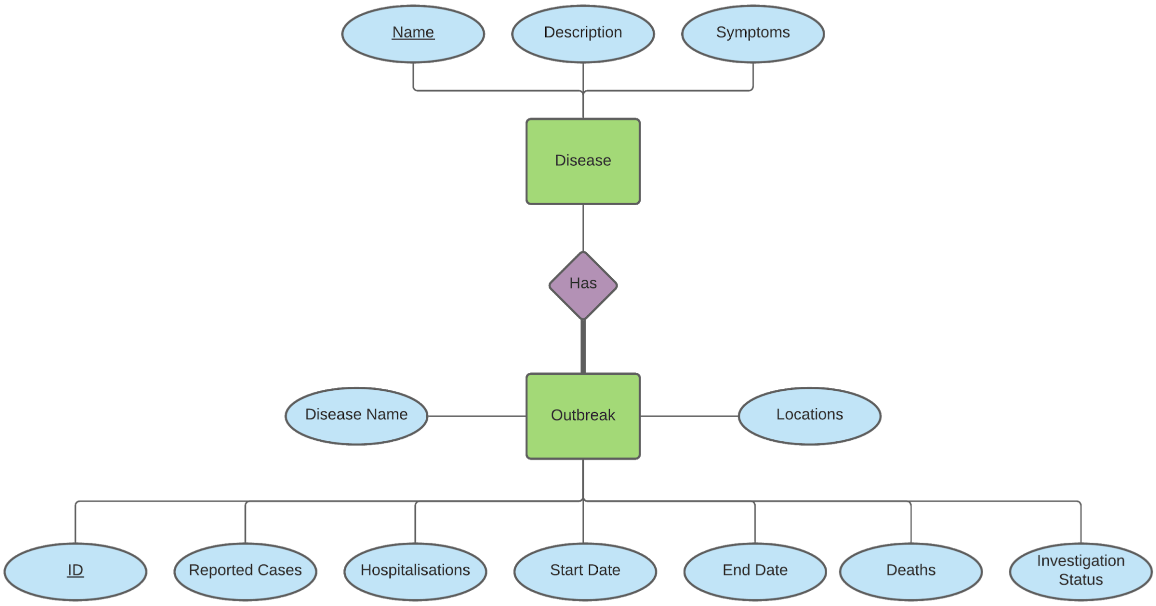


Figure 1. ER Diagram of Outbreaks Database

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.

Language Comparison

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. details our evaluation of the languages we considered.

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:

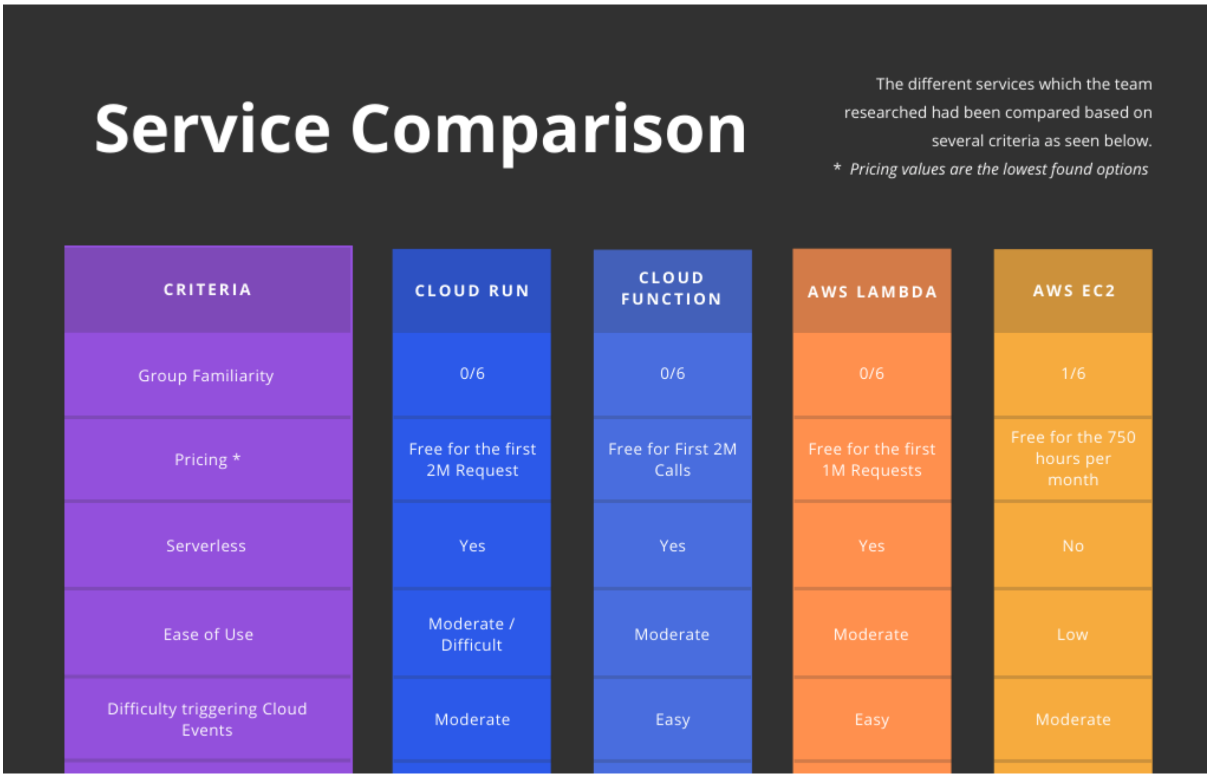
Python Libraries	Strengths	Uses
Scrapy	- Easily extensible - Built-in HTML and CSS data extraction tools - Memory and processing power efficient - Fast - Strong support for web-crawlers	- Will focus on crawling between the various pages of diseases.
BeautifulSoup4	- Shallow learning curve, reducing the time needed for the team to learn the library - Easily extract data out of the requested HTML - Comprehensive documentation - Strong community support	- 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
Selenium	- Automates testing for web applications - Beginner friendly - Has built in tools to mimic human interaction which allows us to have precise control over accessing the different web pages we need	- This will be used adjacent to Scrapy to crawl the CDC outbreak page, given its precise control over a webpage

Figure 3. Python Libraries used with their strengths and uses

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.

Figure 4. shows our evaluation of these options.



Service Comparison

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 Solutions 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
Build a web app			
Web app			✓
HTTP services	✓		
Developing APIs			
API for web & mobile backends	✓		✓
Internal APIs and services	✓		✓
Data Processing	✓		
Automation			
Workflow & orchestration	✓	✓	
Triggered by cloud events		✓	
Connecting Cloud Services		✓	

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	- Mainly charges on writes/reads/deletes - Free Spark Tier is suitable.	Advanced	- Very large data sets - Storing complex, hierarchical data - Advanced querying, sorting, transactions
Firebase Realtime Database	0/5	- Charges only on bandwidth and storage but at a higher rate. - Free Spark Tier is suitable.	Basic	- Data sets that are changed often - Synchronising data - Basic querying - Frequent updates
Google Cloud Storage	0/5	- Charges based on usage. - Not free.	Advanced	- Storing file-like data (e.g. images, binaries, etc.)

Figure 6. Comparison Table of Google Cloud Database Solutions

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. displays a sequence of how an example GET request will look like with our API.

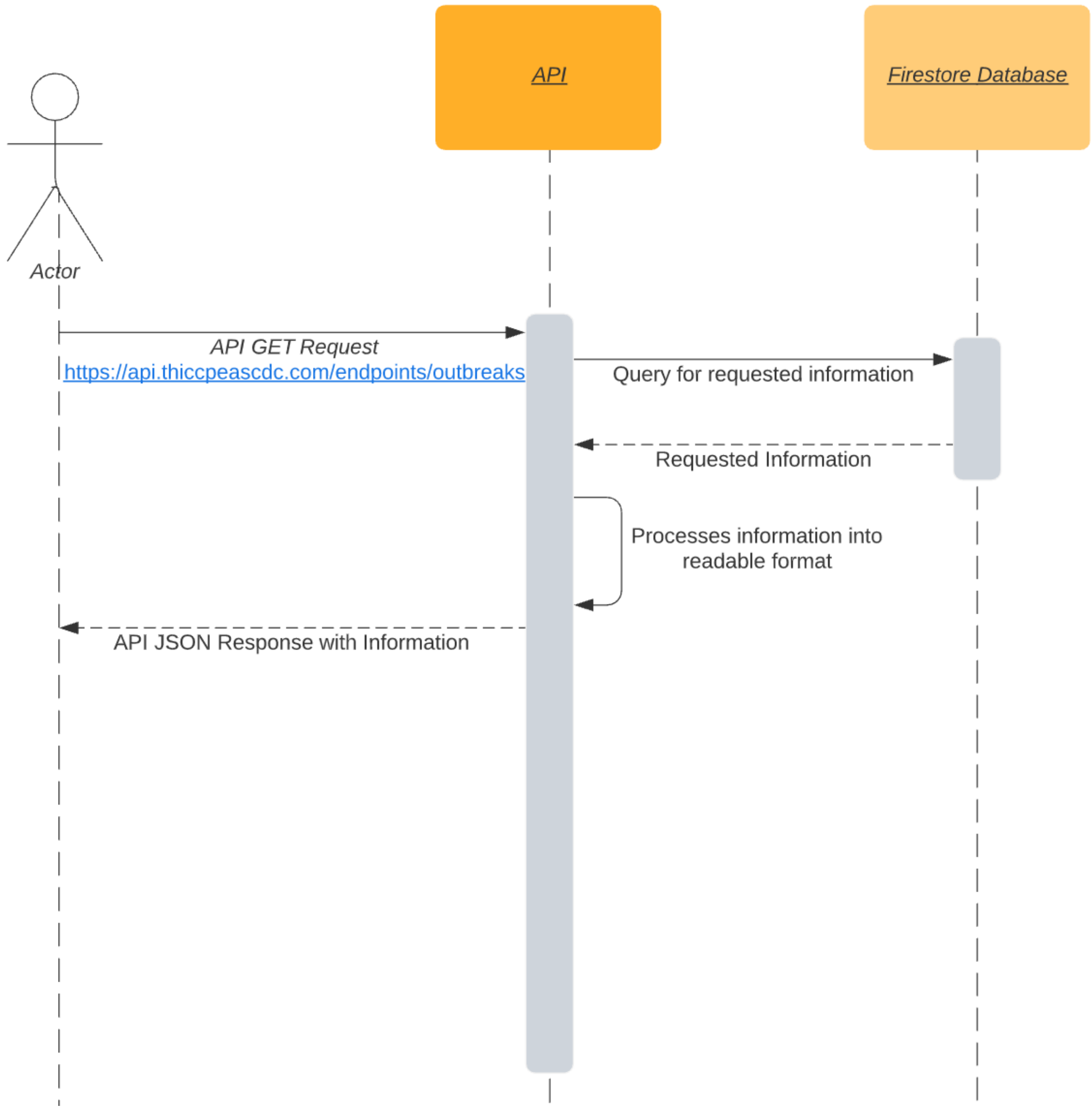


Figure 7. Sequence Diagram of an Example API GET Request

Figure 8. displays the example HTTP GET Requests and endpoints of our API. Please note that the URLs and endpoints are not necessarily representative of the final API.

GET Request	Sample Response

GET Request

```
GET https://www.api.thiccpeascdc/endpoints/outbreaks
content-type: application/json
```

Sample Response


```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 437
ETag: W/"1b5-el1Vi/mcYseZHVVP9lZR83pmuDQ"
Date: Fri, 05 Mar 2021 02:46:03 GMT
Connection: close

{
  "outbreaks": [
    {
      "id": "001",
      "disease_name": "COVID-19",
      "locations": [
        "Lidcome, NSW, Australia"
      ],
      "reported_cases": 17,
      "hospitalisations": 2,
      "deaths": 0,
      "start_date": "07-06-20",
      "end_date": "12-06-20",
      "investigation_status": false
    },
    {
      "id": "002",
      "disease_name": "Listeria",
      "locations": [
        "Colorado, United States of America"
      ],
      "reported_cases": 2,
      "hospitalisations": 1,
      "deaths": 0,
      "start_date": "07-09-20",
      "end_date": "18-09-20",
      "investigation_status": false
    }
  ]
}
```


GET Request

```
GET https://www.api.thiccpeascdc/endpoints?disease=listeria
content-type: application/json
```

Sample Response


```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 484
ETag: W/"1e4-Nabwr/dItEKGoxM7K+NHU4IqgOk"
Date: Fri, 05 Mar 2021 02:54:37 GMT
Connection: close

{
  "name": "Listeria",
  "description": "Listeriosis is a serious infection usually caused by eating food contaminated with the bacterium Listeria monocytogenes. An estimated 1,600 people get listeriosis each year, and about 260 die. The infection is most likely to sicken pregnant women and their newborns, adults aged 65 or older, and people with weakened immune systems.",
  "symptoms": [
    {
      "name": "Fever"
    },
    {
      "name": "Diarrhea"
    },
    {
      "name": "Muscle Aches"
    },
    {
      "name": "Fatigue"
    },
    {
      "name": "Miscarriage"
    }
  ]
}
```

Figure 8. Table of Sample HTTP GET Requests with API

2.2.2 Tech Stack: Language

Figure 9. depicts our comparisons of the possible languages/frameworks of choice for our API.

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[1], 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

[1] Paul Craig, 'Cloud Run vs App Engine: a head-to-head comparison using facts and science', DEV Community, 2020

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:

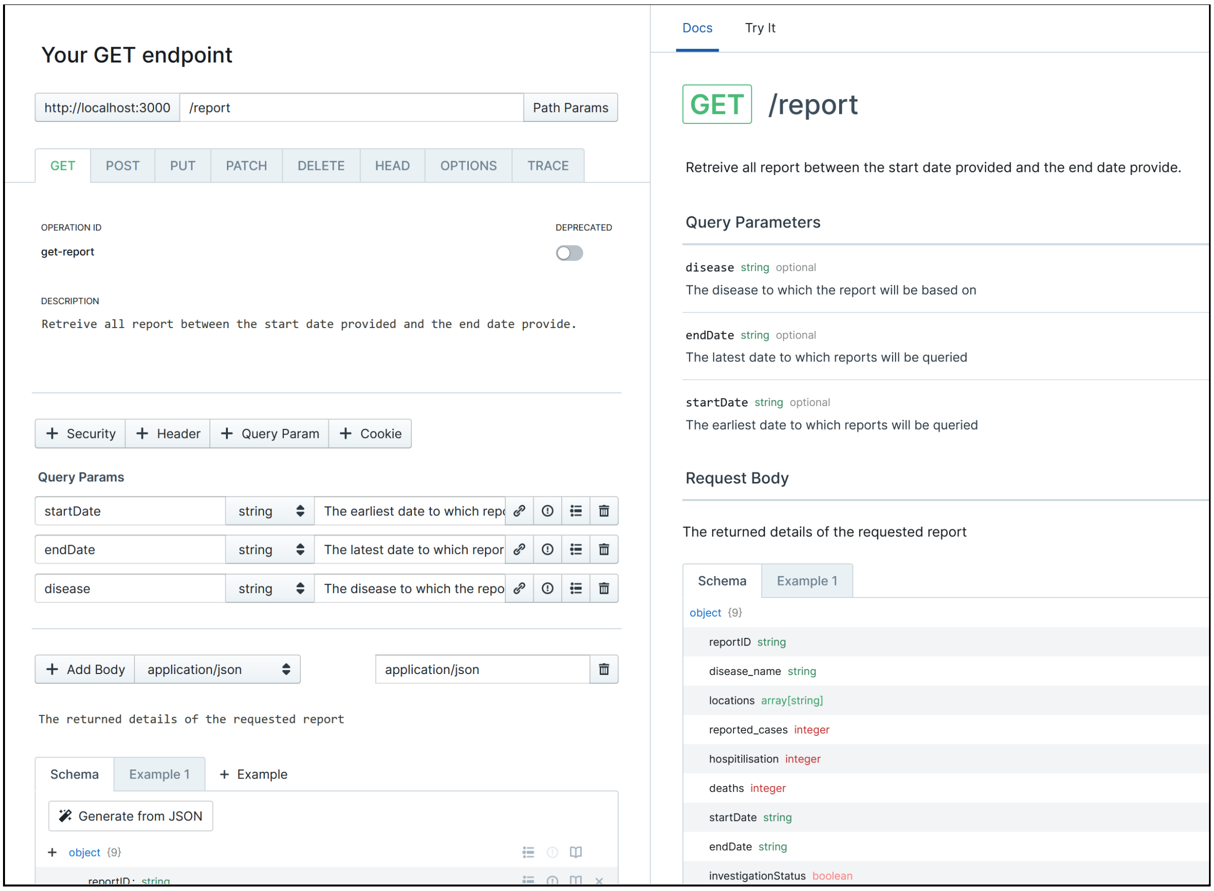


Figure 10. Example Stoplight documentation

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**
- 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. - Spotlight.io has a more easy-to-use GUI than SwaggerHub.	- While collaboration is a possibility, it requires a paid subscription which is not compatible with team budget.

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 economical 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.

2.4.2 Tech Stack: Web Framework and Languages

We considered the following languages for the website:

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 Express.JS, 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.

3.0 Architecture Summary

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.

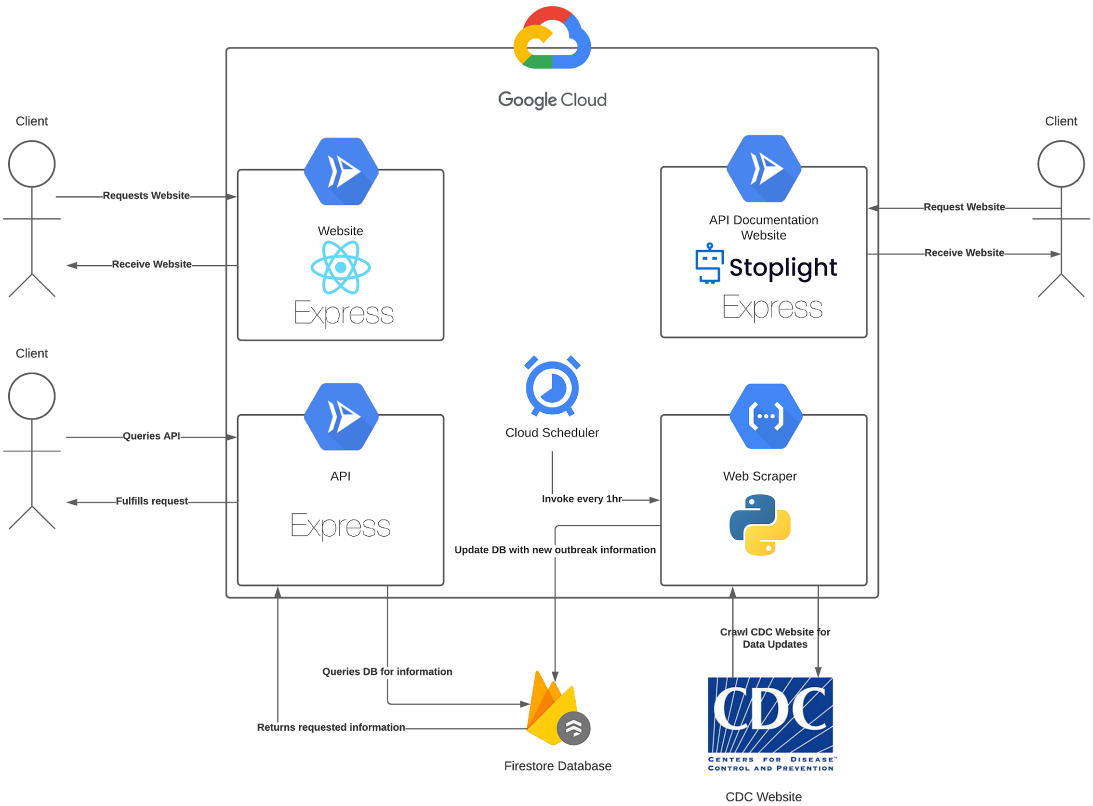


Figure 13. System Architecture Diagram