Design Details Report

Deliverable 1

Group Name:
syntaxError

Project Members:
Neeraj Mirashi
Sohum Jain
Annie Yang
Jordan Lee
Hibah Ansari

1. Task 1 ~ Platform Design

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

1.1 Collaboration Platform: GitHub a version control system

Throughout the duration of the project our team will be using GitHub as a version control system for our API and any relevant documents. GitHub will be used as a means of providing important links for documents, reports, plans etc. Although the use of GitHub was a necessary requirement as per the project-spec, our team believes it truly is the best way of developing this API.

Through the use of GitHub, our team will be practicing the use of 'Feature Branch Workflow'. This essentially means for each different aspect of the API we will create a different branch and work on that branch.

1.1.1 Feature Branch Workflow: Advantages and Disadvantages

Advantages and disadvantages of the 'Feature Branch workflow' are shown in Figure 1.0.

Advantages	**Disadvantages**
• Multiple people work at the same time due to branch encapsulation. • Group members are able to see what other members are working on. • Targeted testing on individual features is much easier • Prevents broken code from entering the main branch	• Merge conflicts are more likely and harder to resolve if commits are not made regularly. • If multiple features are being worked on at the same time, due to the small team size, managing multiple branches can be difficult

Figure 1.0.

Despite some disadvantages, the feature branch workflow is highly beneficial for development and something our group will be implementing all throughout our project.

GitHub provides seamless collaboration without compromising the integrity of our working code and streamlines the iteration process. We believe that GitHub was the right choice for source control.

1.2 Requirements analysis

Before starting the development of our API module, our team believes it is of utmost importance to firstly perform requirements analysis for the given task. Requirements analysis normally composes of *functional* and *non-functional* requirements. During the initial stages of the project our team will primarily be focusing on the functional requirements of the API as things like API performance and response time don't concern us as much at this stage.

1.2.1 Steps for requirements analysis

We have outlined steps below regarding how we will be conducting requirements analysis.

1. Understanding API scope and clearly define/articulate requirements of API.
2. Ensure each team member is familiar with Point 1.
3. Separate functional from non-functional requirements and develop only to functional requirements.
4. Begin development process by identifying API platform.

1.3 Design and Implementation

Our API will be built based on REST (Representational State Transfer) architecture. REST is a series of architectural constraints that define interactions between clients and servers. In REST architecture, clients are required to make requests to retrieve or modify resources, and servers send responses to these requests. For our project, the client will make a request to retrieve or modify a disease report/article or a collection of disease reports/articles. This is created in two parts, a web-scraper which handles aggregation and processing of the disease reports/articles and a front-end API and web-based interface, which handles the clients' requests. Figure 2.0 is a sequence diagram and pictures the initial design of our API:

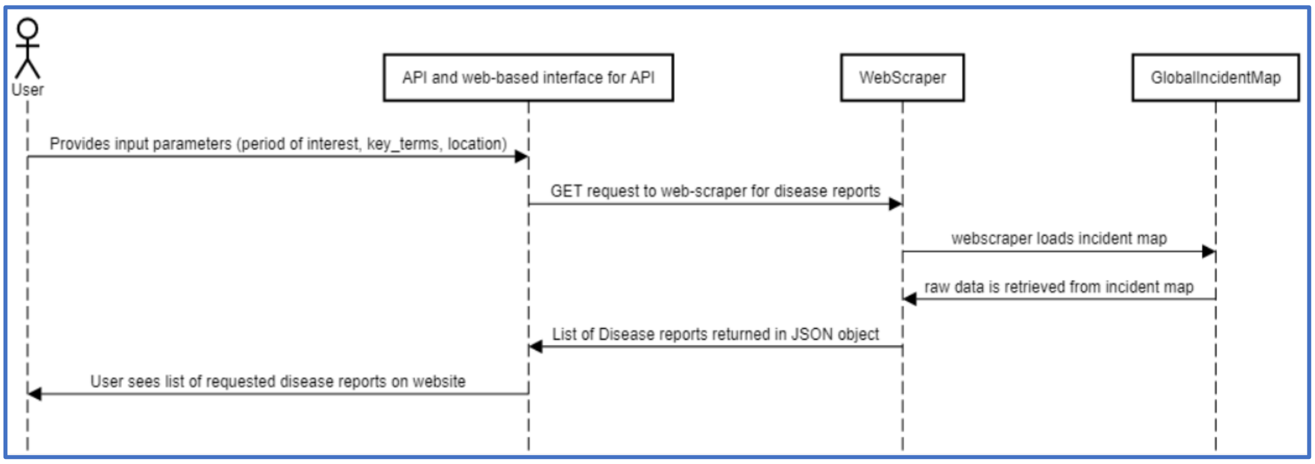

User
API and web-based interface for API
WebScraper
GlobalIncidentMap
Provides input parameters (period of interest, key_terms, location)
GET request to web-scraper for disease reports
webscraper loads incident map
raw data is retrieved from incident map
List of Disease reports returned in JSON object
User sees list of requested disease reports on website

Figure 2.0 – A sequence diagram of what our API does.

1.4 Documentation and Testing

Our API will be documented based on OpenAPI specifications. The Swagger Editor is an Open-Source editor that allows us to design and document our RESTful API in the swagger Specification. Our team will use Swagger editor to create our API Specification based on OpenAPI spec syntax guidelines. Swagger Editor will then allow us to generate a python-flask server and client and test it using the Swagger web UI (User Interface). We will then convert into a Postman Collection for further testing.

1.4 API as a web-service

Our API will be created using Flask and Deployed to AWS. After creating a basic front-end, we will use the route Flask endpoint to register our functions with the given URLs and is then deployed to AWS.

2. Task 2 ~ Parameters and Endpoints of API

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

2.1. Retrieving from the user

To retrieve the relevant search data from the user, we will be using a HTTPS GET request via an HTML form. The parameters required for this method include key terms, location, and period of interest. A successful request will return the response from the GET /results/ request to the server. An invalid request will prompt an error pop-up box indicating there has been an error and the user must try again.

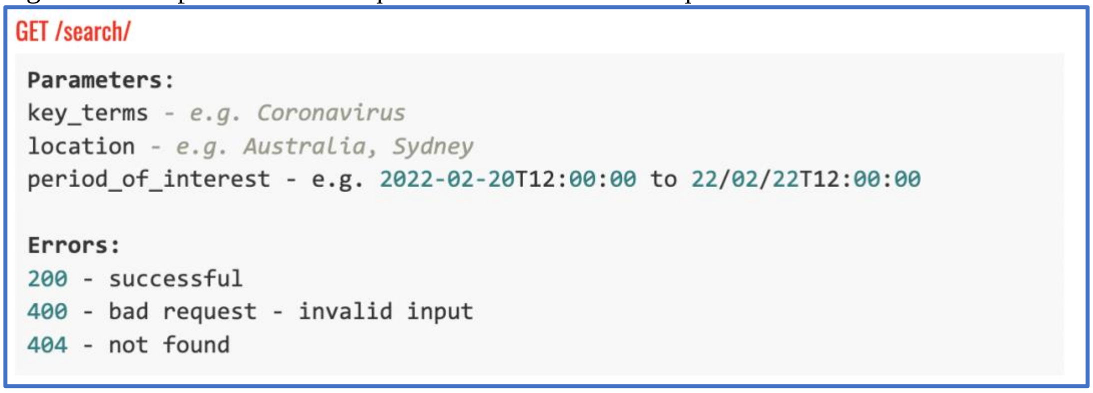
2.2. Retrieving from the website

Once the user has submitted the request, our API will search the Global Incident Outbreaks website, parse the relevant details, and return the response of that GET request to the user. The response JSON will be a list of report objects. Each report object will include the report id, URL, date, title, disease type and a list of details. Each detail object will include the type (infection or death, date, location, and number).

2.3. Possible Interactions

2.3.1 Example - GET request

Figure 3.0 is a possible GET request and shows the subsequent errors that could occur.

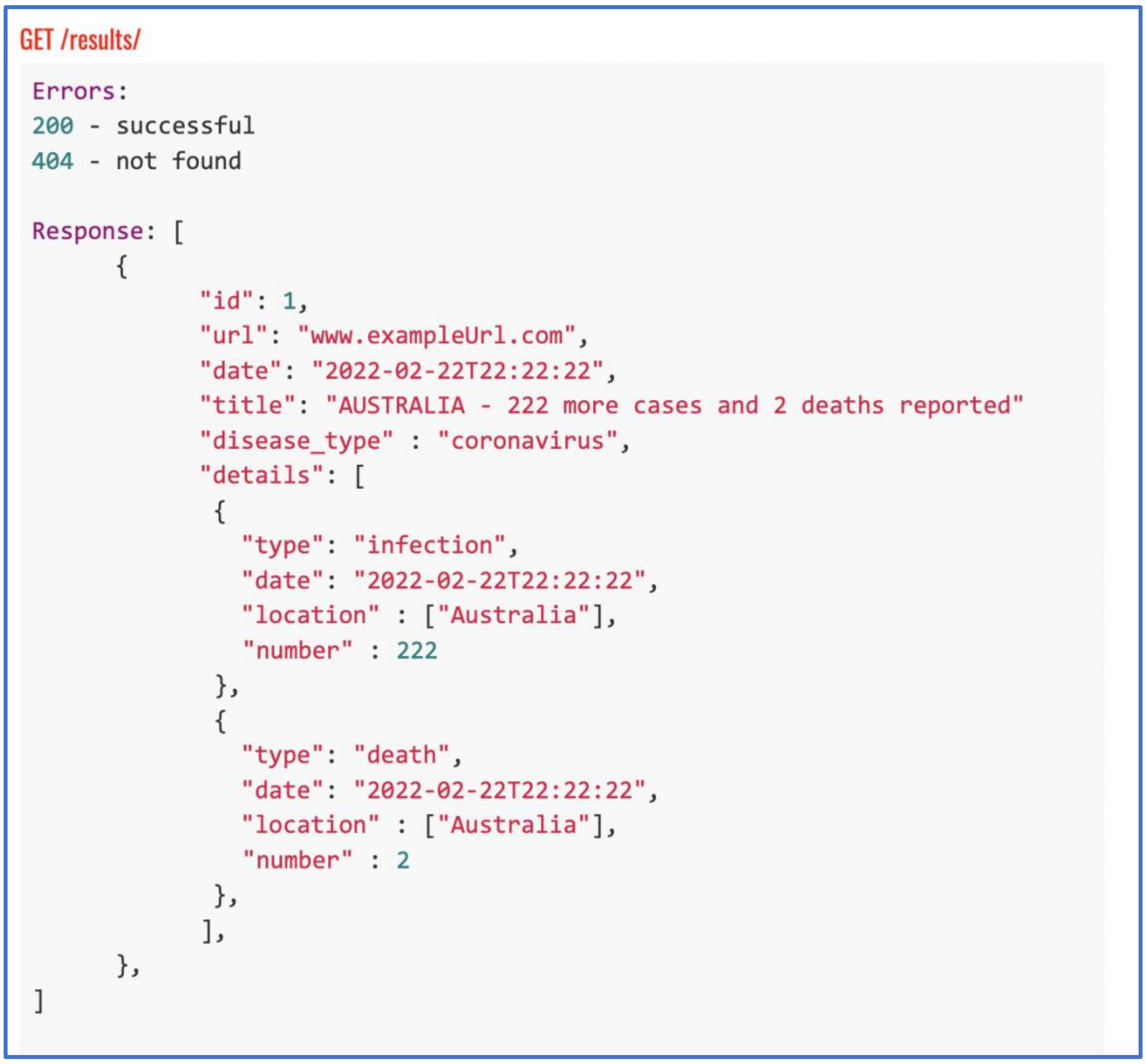


```
GET /search/

Parameters:
key_terms - e.g. Coronavirus
location - e.g. Australia, Sydney
period_of_interest - e.g. 2022-02-20T12:00:00 to 22/02/22T12:00:00

Errors:
200 - successful
400 - bad request - invalid input
404 - not found
```

Figure 3.0.

2.3.2 Example - Response returns a list of all results

Figure 4.0 below is a possible response to a GET request. The response contains a list of results.



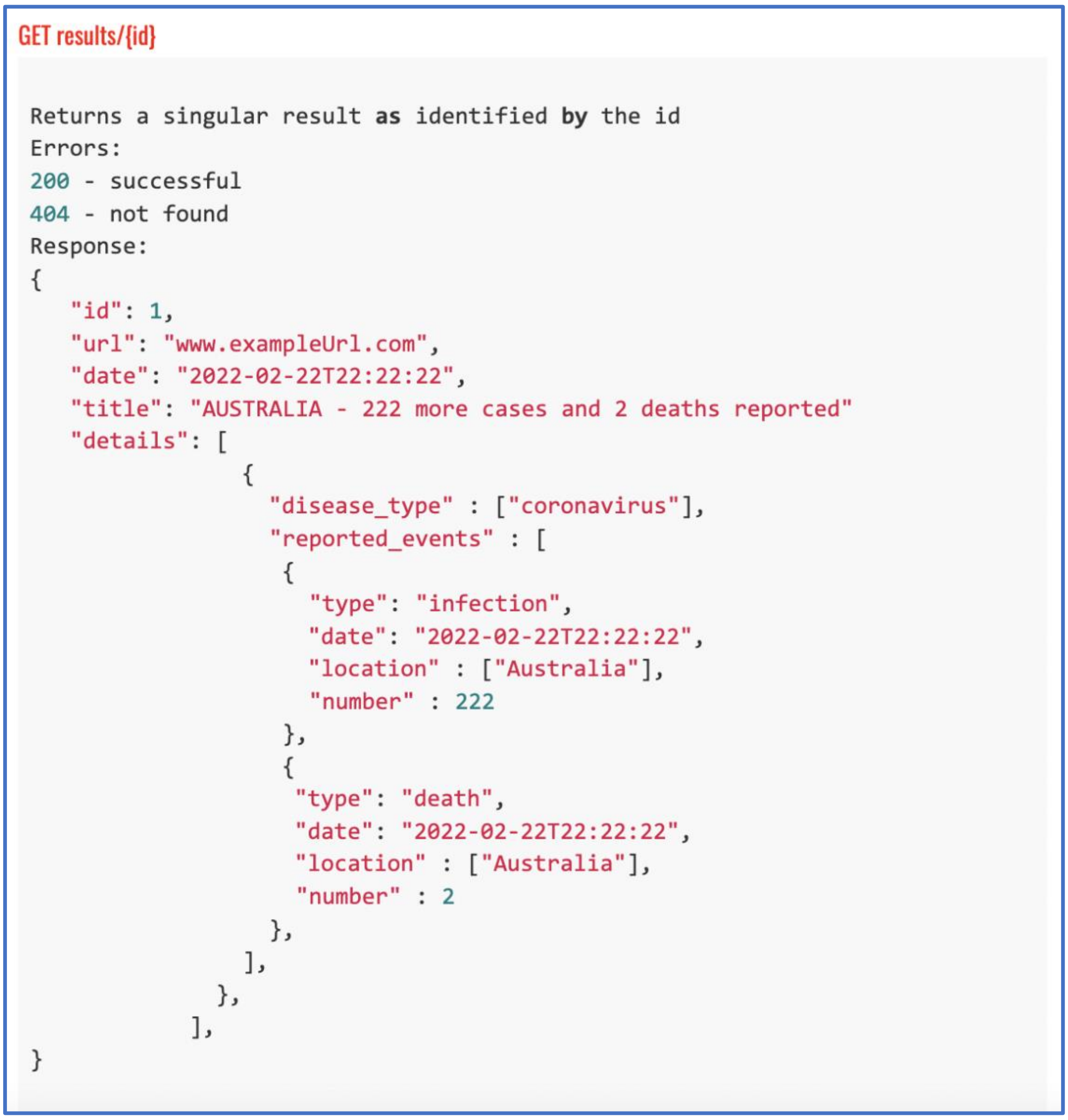
GET /results/

```
Errors:
200 - successful
404 - not found

Response: [
      {
            "id": 1,
            "url": "www.exampleUrl.com",
            "date": "2022-02-22T22:22:22",
            "title": "AUSTRALIA - 222 more cases and 2 deaths reported"
            "disease_type" : "coronavirus",
            "details": [
             {
               "type": "infection",
               "date": "2022-02-22T22:22:22",
               "location" : ["Australia"],
               "number" : 222
             },
             {
               "type": "death",
               "date": "2022-02-22T22:22:22",
               "location" : ["Australia"],
               "number" : 2
             },
            ],
      },
]
```

Figure 4.0.

2.3.3. Example - Response returns a single result

Figure 5.0 is a possible response to a GET request. The response contains a single result.



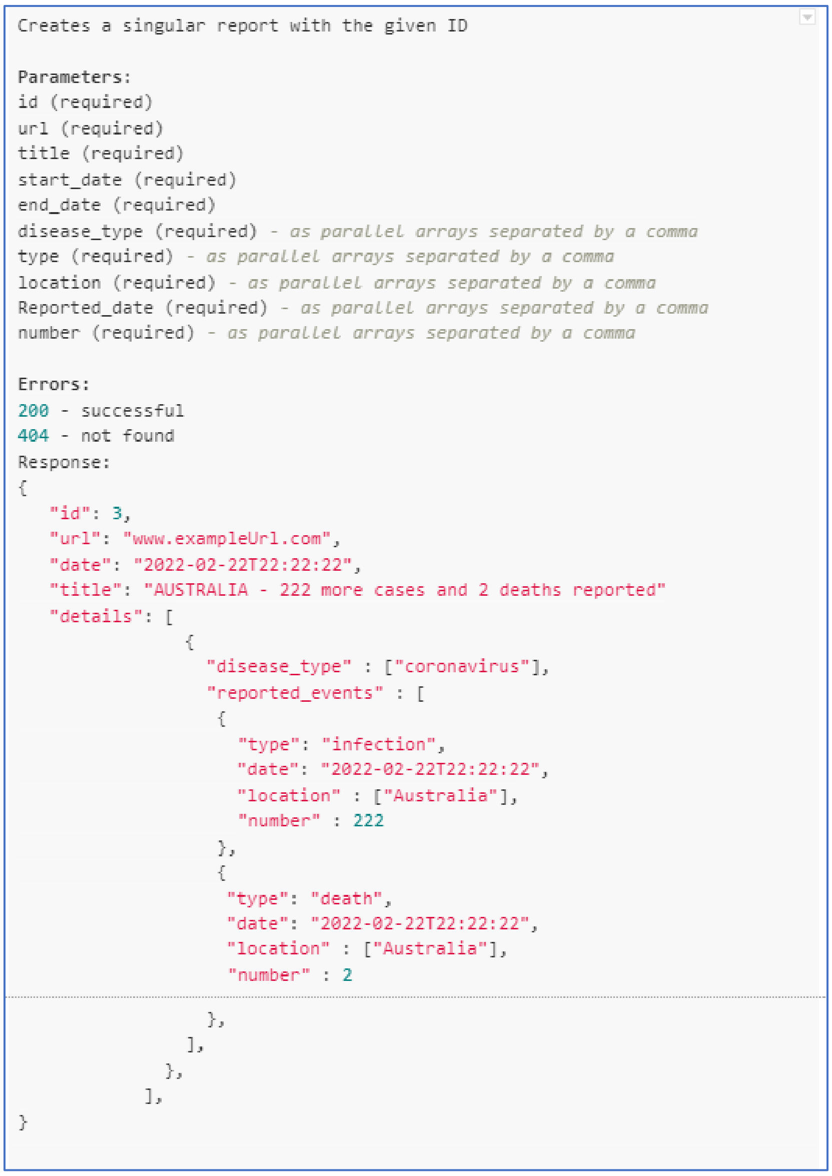
GET results/{id}

```
Returns a singular result as identified by the id
Errors:
200 - successful
404 - not found
Response:
{
   "id": 1,
   "url": "www.exampleUrl.com",
   "date": "2022-02-22T22:22:22",
   "title": "AUSTRALIA - 222 more cases and 2 deaths reported"
   "details": [
                {
                  "disease_type" : ["coronavirus"],
                  "reported_events" : [
                   {
                     "type": "infection",
                     "date": "2022-02-22T22:22:22",
                     "location" : ["Australia"],
                     "number" : 222
                   },
                   {
                    "type": "death",
                    "date": "2022-02-22T22:22:22",
                    "location" : ["Australia"],
                    "number" : 2
                  },
                ],
              },
            ],
}
```

Figure 5.0.

2.3.4. Example – Post request for single report

Figure 6.0 is a possible response to a POST request. The response generates a report based on the user-inputted parameters.

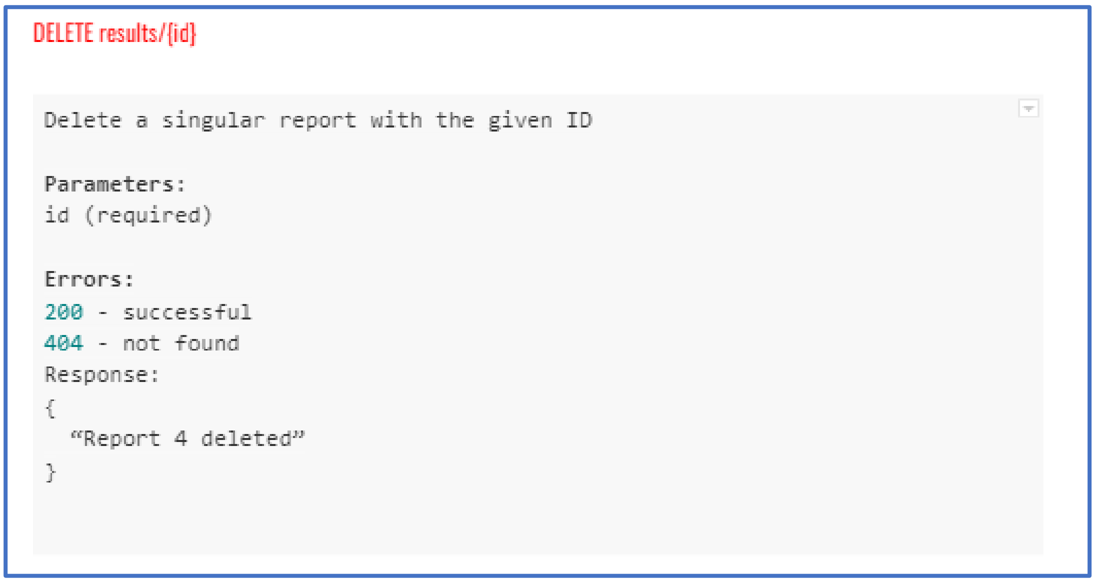


```
Creates a singular report with the given ID

Parameters:
id (required)
url (required)
title (required)
start_date (required)
end_date (required)
disease_type (required) - as parallel arrays separated by a comma
type (required) - as parallel arrays separated by a comma
location (required) - as parallel arrays separated by a comma
Reported_date (required) - as parallel arrays separated by a comma
number (required) - as parallel arrays separated by a comma

Errors:
200 - successful
404 - not found
Response:
{
   "id": 3,
   "url": "www.exampleUrl.com",
   "date": "2022-02-22T22:22:22",
   "title": "AUSTRALIA - 222 more cases and 2 deaths reported"
   "details": [
                {
                  "disease_type" : ["coronavirus"],
                  "reported_events" : [
                   {
                     "type": "infection",
                     "date": "2022-02-22T22:22:22",
                     "location" : ["Australia"],
                     "number" : 222
                   },
                   {
                    "type": "death",
                    "date": "2022-02-22T22:22:22",
                    "location" : ["Australia"],
                    "number" : 2
                  },
                ],
              },
            ],
}
```

Figure 6.0.

2.3.5. Example – Deletes Single Report

Possible Response to a DELETE request. The response is a message verifying report is deleted



DELETE results/{id}

```
Delete a singular report with the given ID

Parameters:
id (required)

Errors:
200 - successful
404 - not found
Response:
{
  "Report 4 deleted"
}
```

Figure 7.0.

3. Task 3 ~ API Implementation

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

3.1. Frontend

3.1.1 Front-end languages

Our team will use React to develop the front-end of our application due to having an exposure to it from previous engineering workshops. React is well documented, and allows direct API implementation within the front-end, expediting the development process. We will also use common front-end languages such as JavaScript, HTML and CSS.

3.1.2 Frontend languages comparison

We compared three popular frontend frameworks: React, Vue and Angular. From the table below, it is evident that the group had more experience in React compared to the other two. In addition, we found React and Vue to be more beginner friendly while Angular was more difficult to get started with since it is based on Typescript. Thus, based on our expertise and analysing our options, we decided to go with React.

	React	Vue	Angular
Group expertise	✓	✗	✗
Component reusability	✓	✓	✓
Beginner friendly and well-documented	✓	✓	✗

Figure 8.0.

3.2. Backend

3.2.1 Backend languages

Python will be used as the primary language for our development as it is well-documented and simple to use. Based on an initial survey of languages used among our team, Python was found to be the most used language among our team members for backend programming. Further, our team considered the availability of Python libraries such as Selenium, Pandas and Scrapy which assist with the implementation of a web scraper as well as visualizing data in our project.

3.2.2 web-frameworks comparison

Our team compared the 3 most popular web-frameworks and summarized our findings into the table below. Our team will use Flask API to develop our web API. It is fast and provides built-in support for Swagger which will be used to generate API documentation.

	Flask	Django	FastAPI
API Support	✓	✗	✓
RESTful URL dispatcher	✓	✗	✓
Supports multiple databases	✓	✗	✓
Beginner friendly and well-documented	✓	✗	✗

Figure 9.0.

From this analysis, Flask fit perfectly with our requirements and hence we finalised it for this project.

3.3 Databases

For data storage, our team decided to use MongoDB as our database system since it is highly flexible and can be used to develop scalable applications. The primary benefit of using mongo DB is its storage of data in JSON files in a schema less database. As compared to RDBMS based databases such as PostgreSQL, mongo DB provides greater flexibility, allowing for data-retrieval for any data type, as well as making no changes to data. It also leaves the raw data untouched and unformatted which allows for better data visualization/representation.

3.4. Development and Deployment Environment

3.4.1 Operating Systems

As we are working from our local machines, our development environment will be MacOS and Windows, keeping all stacks and technologies compatible. This was preferred over Linux as team members do not need to SSH into CSE servers to access the environment.

3.4.2 IDEs

For the development environment, we have decided to use VSCode as it is a light-weight code editor with a simple, user-friendly interface. VSCode provides git integration as well as quick code-build-debug cycle. Furthermore, VScode offers VSCode Live Share which enables collaboration on code which will be a useful feature.

3.4.3 Web-browsers

Our product will be platform-agnostic and compatible with all browsers. Testing and development will be done on Opera, Chrome, Safari, Edge and Firefox.

3.4.3 Hosting

AWS is a cloud hosting platform that allows users to host their web applications. We chose this as our cloud platform as it is beginner-friendly and easy to use. We compared three different hosting platforms as shown in the table below. For our requirements, we found AWS to be the best fit.

	AWS	Azure	Google Cloud
Testing Features	✓	✓	✓
Beginner-friendly	✓	✗	✓
Task-related features	✓	✗	✗
Virtual private cloud	✓	✗	✓

Figure 10.0.

3.5. Specific Libraries Used

3.5.1 Testing libraries

Selenium WebDriver is a collection of open-source APIs which are used to automate interaction with a web application. This is useful for testing purposes, ensuring that all components of our front-end are working, as well as to ensure that appropriate results are scraped from the Global Incident map website.

Our team will use Postman for the testing of our API. Postman allows teams to collaborate when writing test cases, as well as providing an environment to store, write and run the test cases. Postman was preferred over Insomnia since it provides documentation services that the latter doesn't offer.

3.5.2 Parsing Libraries

Beautiful Soup is a Python library for pulling data out of HTML and XML files. Our team will use Beautiful Soup to process and format the raw HTML data as well as parse this data to extract the information into a JSON package for further usage**.** Beautiful Soup will be used for parsing HTML responses in Scrapy call-backs.

3.5.3 Scraping Libraries

Our team decided to use Scrapy to extract data from the outbreak website. This is because Scrapy can extract data in JSON, is faster than alternative scrapers such as Selenium and BS4 as well as allowing easy post-production of large volumes of data with little memory and CPU overhead.

3.6. Software Architecture Summary

From the diagram below, we can see that the web application supports multiple browsers (Opera, Chrome, Edge, Firefox, Safari). The client-side will interact with the server via HTTPS GET Requests which will be reciprocated via HTML responses. Within the server side, we have the frontend and backend implementation as well as the API and database control. The frontend will be implemented in React, Javascript, HTML and CSS. The backend will be implemented in Python and FlaskAPI along with Python libraries mentioned in Section 3.3. For data storage, we have opted for MongoDB which will be used to store and retrieve API results.

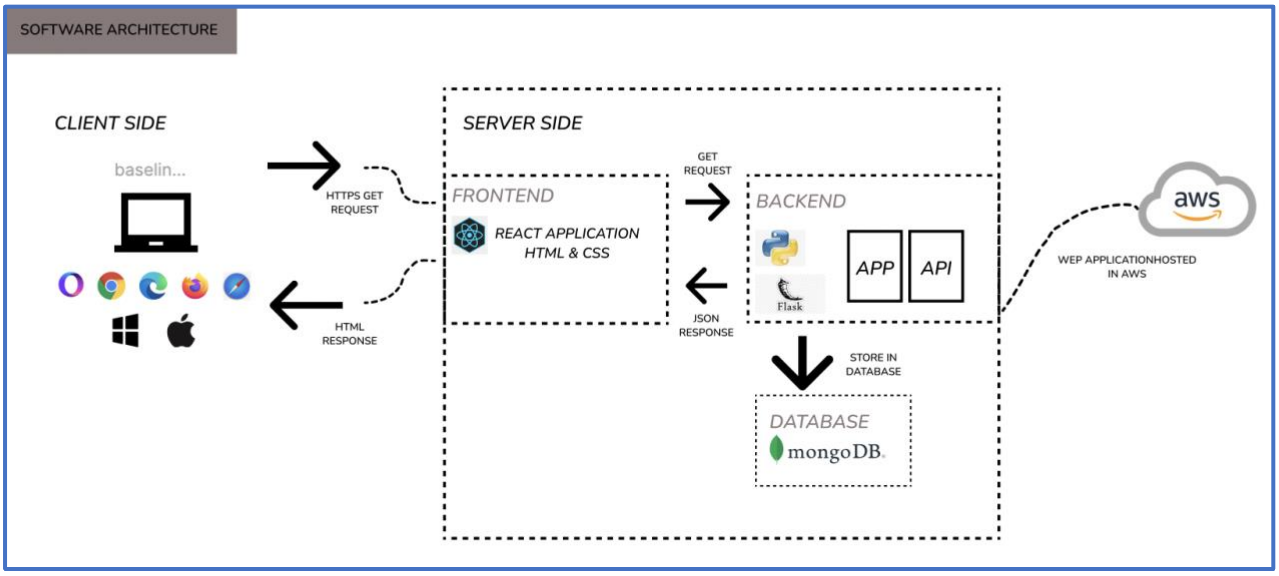


Figure 11.0.

4. Task 4 ~ Above and Beyond

4.1. Selenium Search

Our team began experimenting with implementations of the above report and decided to trial using Selenium to narrow down our search-results. Below is Footage of our team using an automated search for Coronavirus in Australia, narrowing down the breadth of results we would potentially have to parse.
https://github.com/Neeraj-A-M/SENG3011_syntaxError/blob/main/selenium-footage.mp4

4.2. Geolocation Taxonomies

Our team started researching into geolocation taxonomies and found the Geonames API. The Geonames geographical database covers all countries, and allows for full-text search, geocoding, and reverse geocoding. Our team plans to use geocoding followed by reverse geocoding to get the exact locations the client specified. This process will allow our team to accurately detect whether a city is within a state (Sydney is inside NSW).

4.3. Data Visualization

Our team started researching into methods of Data representation for our API. Integrating our API with the Google Maps API allows our team to visualize of location-based data. Every report selected can be displayed as a heat zone in a central map.