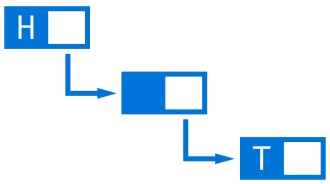
Deliverable 1



Team: Linked List

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

0. Table of contents

0. Table of contents **2**

1. Describe how you intend to develop the API module and provide the ability to run it in Web service mode **3**

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

2. Discuss your current thinking about how parameters can be passed to your module and how results are collected. Show an example of a possible interaction. (e.g.- sample HTTP calls with URL and parameters) **5**

1. Overview 5
2. Headers 6
3. Parameters 7
4. Example of Using the API 9
    I. As a developer 9
    II. As a frontend user 11
5. Handling Errors 11

3. Present and justify implementation language, development and deployment environment (e.g. Linux, Windows) and specific libraries that you plan to use. **13**

I. Backend Language 13
II. Scraper 14
III. Deployment 15
IV. Database 16
V. Frontend 17
VI. Development 18
VII. API documentation 19
VIII. Testing 19

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

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

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

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

1. Data Collation

We will be using the BeautifulSoup4 Python library to build a Python webscraper to extract data out of www.cdc.com.gov. The scraper will automatically follow relevant links on https://www.cdc.gov/outbreaks/ using a depth first search approach and collate all information regarding a particular disease together into one object. A depth first search would allow us to collate all relevant information with regards to a specific disease report. The reason for choosing beautiful soup over other libraries is illustrated in Q3.

To ensure our data is up to date, we will use Google Cloud Platform's Cloud scheduler to schedule the webscraper to scrape through the website when a query is sent to our API service, unless the webscraper is activated within the same week. At a minimum, a weekly schedule ensures that our database contains the most relevant information, but if our API service is not being used then it will not activate to ensure that we do not waste cost running our service.

2. API Architecture Planning and Data Restructure

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

We further want to enhance the user experience with additional endpoints other than the search endpoint. For example, we want the API user to query a disease report directly if it already knows it's ID.

We will be populating the collected data to firestore. We will enable the firestore database through firestores GUI, then build endpoints for our API by using cloud functions. The collected data will be converted to JSON then populated into Cloud Firestore using POST method. The screenshot below is an example of how we are planning on populating data into Cloud Firestore with cloud functions.

```
app.post('url', async(req,res) => {
    try{
        const example = {
            attribute: req.body['exmaple'],
            attribute: req.body['exmaple'],
            attribute: req.body['exmaple']
        }
        //Insert the object
        await db.collection('collected_data').doc().create(exmaple)
        return res.status(200).send("target url name")
    }
})
```

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

3. Making the data available

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

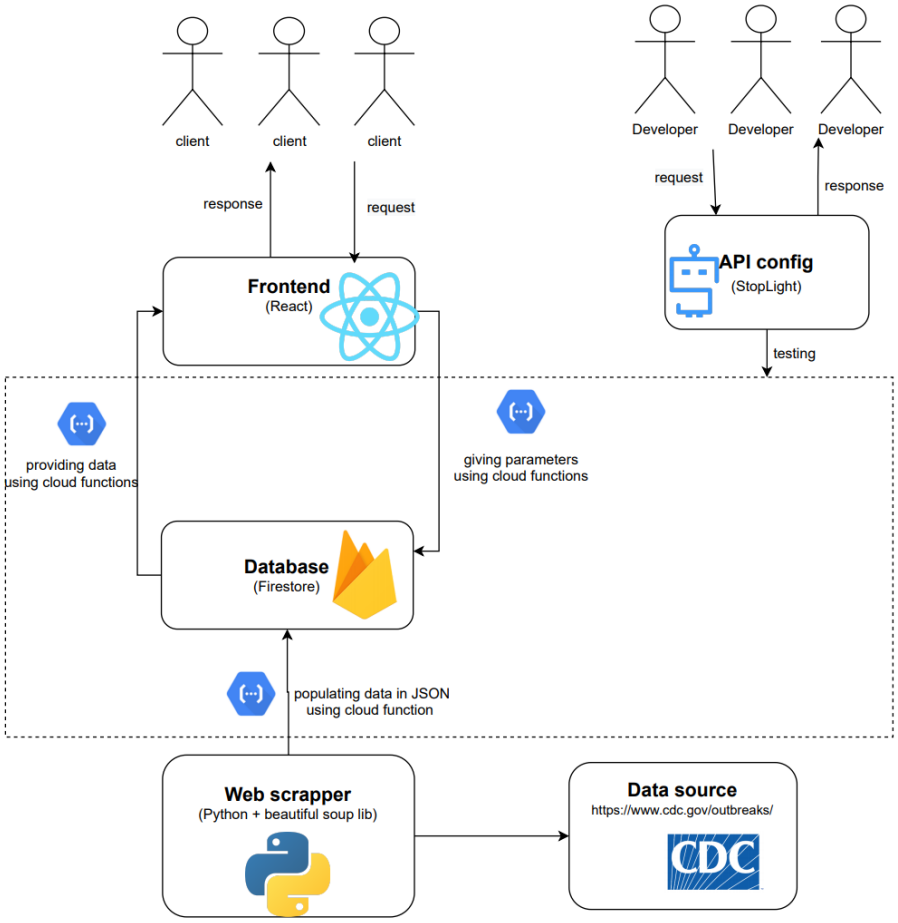
We will be updating the database using PUT cloud functions to make changes to objects that are already in the database. We will be updating it either on a regular basis (every 10 mins) or checking the last modified date to keep it up-to-date. We will also be building multiple GET methods with different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

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

4. Testing API

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

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



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

a. Overview

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

b. Headers

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

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

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

Now onto the response headers.

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

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

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

c. Parameters

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

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

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

	JSON	**PLAIN TEXT**
Pros	- Easier to use in functions, which you can just get the value to the given key - Is essentially plain text, however a pre built library can be used to operate thus - Has more pre built libraries which are reliable - Whole team has very in depth experience dealing with JSON	- Easier to write - No complexity to it, thus you can add your own structures
Cons	- Can be hard to read at times when there is a lot of content	- Less library support - Need to self write functions to get

		parameters - Can be hard to read at times when there is a lot of content

Other considerations we made were XML, YAML and CSV. However in terms of the team's overall experience and pre built library support. JSON proved to be the best decision.

d. Example of Using the API

I. As a developer

Below is an example of what a developer would do in a Python3 program or shell. For the developer to use our API, they need to generate an API key and a JSON.

The example below shows how the JSON param is generated with the use of JSON and datetime libraries, followed by the creation of the JSON string and then used in the API by curling our mock web domain with a get command.

To generate an API key, the developer must visit our *for developers section* on our front end to enable them to use our API

```
# IF USING THE PYTHON SHELL IN TERMINAL CALL python3 ELSE LEAVE THIS OUT
python3

# IMPORTING LIBRARIES:
import json
from datetime import datetime

# SETTING UP THE DATE TIME VARIABLES
b = datetime(2020, 01, 03, 00, 00, 00, 000000).strftime("%m/%d/%Y, %H:%M:%S")
c = datetime(2020, 01, 15, 00, 00, 00, 000000).strftime("%m/%d/%Y, %H:%M:%S")

# PYTHON OBJECT (DICT):
params = {
   "location" : [“China”, “Japan", “Wuhan"],
   "periodOfInterest" : [b,c],
   "keyTerms" : ["Coronavirus", "Novel Coronavirus"]
```

```
}


# CONVERT INTO JSON
y = json.dumps(params)


# IF USING THE PYTHON SHELL IN TERMINAL USE THIS COMMAND.
curl https://chicken.com/api/get_json?params=y&api_key=EXAMPLEKEY
```

Below is an example showing how a possible GET request from API endpoint may look like:

`GET/reports?start_date=2020-09-01T09:00:00&end_date=2021-5-03T09:00:00&key_term=corona&location=china`

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

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

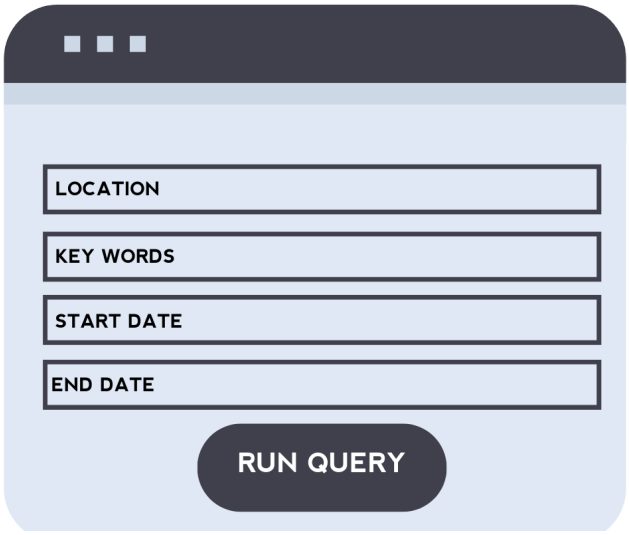
```
{
  "url": "https://www.who.int/csr/don/17-january-2020-novel-coronavirus-japan-ex-china/en/",
  "date_of_publication": "2020-01-17 xx:xx:xx",
  "headline": "Novel Coronavirus – Japan (ex-China)",
  "main_text": "On 15 January 2020, the Min
istry of Health, Labour and Welfare, Japan (MHLW) reported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

II. As a frontend user

As a user from the front end, they will be displayed with a HTML page, the link of the given html page will be as follows.

```
https://chicken.com/api/get_json?params=y&api_key=EXAMPLEKEY
```

The params variable will be created like the one above in the developer example when the user clicks the query button on *the simplified wireframe.* The API key attached to the query will be a generic API key which is for front end usage only.



e. Handling Errors

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

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

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

HTTP Response Status Code	Scenario	Example Error Message
`400 BAD REQUEST`	- When the request is invalid, e.g. invalid or missing timestamp	`400 'key_terms' must not be empty`
`401 UNAUTHORIZED`	- When the user does not provide a valid api key	`401 Please provide a valid api key`
`404 NOT FOUND`	- When the endpoint cannot be found on our API service	`404 '/foo/bar' does not exist`
`500 INTERNAL SERVER ERROR`	- When an uncaught error/exception is thrown in our API Service, e.g. null pointer exceptions	`500 Internal server error`
`501 NOT IMPLEMENTED`	- When an endpoint under development is called. This error will not be thrown in our final product.	`501 '/foo/bar' is still in development`
`502 BAD GATEWAY`	- When Firebase returns an invalid response (or a response the our API service cannot interpret)	`502 Server returned invalid response`
`504 GATEWAY TIMEOUT`	- When Firebase is unavailable - When Firebase is taking too long to respond	`504 Server did not respond in time`

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

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

Overview

Our team decided to implement the web scraper in python along with the beautiful soup library. We will be developing our API serverlessly with the help of firebase - a google cloud platform. Our data will be populating into firstore as our database. We will have an API to establish connections between clients and database. It will be documented with stoplight, and having its endpoints built by cloud functions. Those cloud functions will upload the collected data to firestore, as well as respond to clients through the frontend. Frontend will be built using React framework, CSS and HTML5.
Our design choices were justified by comparing with other possible tools that can be used during development. Those justifications will be thoroughly presented below.

I. Backend Language

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

Python	Java	Conclusion
Python is a **scripting language**. This means it is: ◍ more suitable for developing small applications ◍ extremely suitable for prototyping in the early stages of application development ● harder to develop additional features once	Java is a **verbose and compiled language that enforces Object Oriented Programming**. This means it is: ◍ more suitable for creating stable products ◍ easier to build additional features on top of MVP since OOP encourages future proofing through its design patterns	Due to the small time frame of our development (8 weeks), we would want to use **Python** to quickly develop a MVP.

MVP has been made unless OOP has been used	requires more time to develop an MVP requires the developer to understand OOP principles	
All of the developers on our team already know Python	Only two developers on our team know Java, and only at a beginner level. However, we are open to learning a new language if it results in a better product.	We would want to use **Python** if Java does not provide any outstanding benefits to reduce development time. We need to complete the API before D2. The time is less than 3 weeks, and we cannot afford the cost of learning Java.

II. Scraper

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

We finalised our decision on BeautifulSoup4. BeautifulSoup4 is a beginner-friendly language that has extensive documentation. Compared to Python Requests, it is easier to extract HTML elements due to its advanced built-in functions, rather than using regex. Selenium as another beginner-friendly language is considered a good choice. The advantage of it is that it can run Javascript, however in this case it is not necessarily needed since the content is loaded statically from the base url. Also using Selenium needs a browser running which occupies more resources and makes it slower.

Scraping tools	Ease of use	Documentation	Speed
Beautiful Soup	Beginner-friendly	Extensive documentation	slow

LXML	not beginner-friendly	Documentation is not beginner-friendly	fast
Python Requests	Beginner-friendly	good documentation	slow
Scrapy	not beginner-friendly	Excellent documentation	fast
Selenium	Beginner-friendly	Extensive documentation	slow
MechanicalSoup	not beginner-friendly	unmaintained for several years, does not support python3	medium
Urllib	not beginner-friendly, bad interface	Documentation is not as easy as other choices	medium

III. Deployment

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

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

Firebase(serverless)	**AWS RDS(serverless)**	**Server-based backend(eg. Python flask)**	**Conclusion**
Do not need to run the server constantly	Do not need to run the server constantly	Have full control to our server Free to choose	

		team's preferred language Have to run the server constantly	
Firebase can be set up very easily. Because Firebase Management SDK supports python. Only use the SDK reasonably to use firebase on python	Higher learning costs, because AWS has many modules to deal with different problems. We need to learn the corresponding modules for each requirement	Members have some flask+python experience But if you need to deal with different problems, you may even need to have a deeper understanding of related librariesy	In terms of learning costs, firebase is almost the best choice. Because he can let us develop our project quickly
Due to the serverless structure, there may be some delays when a "cold start" occurs, but in most cases it is okay.	The optimized database provides better performance.	This is based on the servers we can rent, but in fact, due to insufficient scalability, it is likely to exceed the access limit.	AWS may be the best choice, but firebase is close behind.
Excellent synergy brought by serverless architecture. This reduces the reliance on hardware and operating system	Excellent synergy brought by serverless architecture. This reduces the reliance on hardware and operating system	Need to depend on the operating system.	At this point, AWS and firebase have gained significant advantages.
Has a 90 days free trial with unlimited access to all functionalities. If we go over time, we need to pay a fixed amount of money regularly.	Free trial for 750 hours, however team members might accidentally leave an instance on and thus use up a big chunk of our time allowance.If we go over time, we need to pay a fixed amount of money regularly.	We need to determine the load of the server in advance, and rent a suitable server, which may be very expensive. The fee depends on the usage of the server instead of paying a fixed amount regularly.	Firebase is one point ahead in the price war, because we cannot estimate whether our current load will exceed its free limit, so we might as well try firebase first, and it is cheaper than AWS.

IV. Database

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

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

V. Frontend

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

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

VI. Development

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

	Windows	Linux	Conclusion
Development environment	Everyone installed windows	Can only be developed on the school's vlab	This tends to be windows
Operating environment **(Compatibility)**	Need to use a virtual machine to determine whether it can run on	Can run smoothly in school. But it needs to be tested in the windows environment.	We can use the school's vlab for testing Linux and use **VMware** test other

	the school's Linux system. There are some tutorials on the Internet that can use **Hyper-V or VMware**		environments, so it's no harm at all.
Team experience	All team members have experience		

VII. API documentation

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

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

VIII. Testing

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

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

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