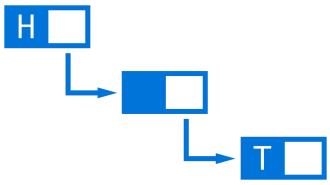
Deliverable 1



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

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.

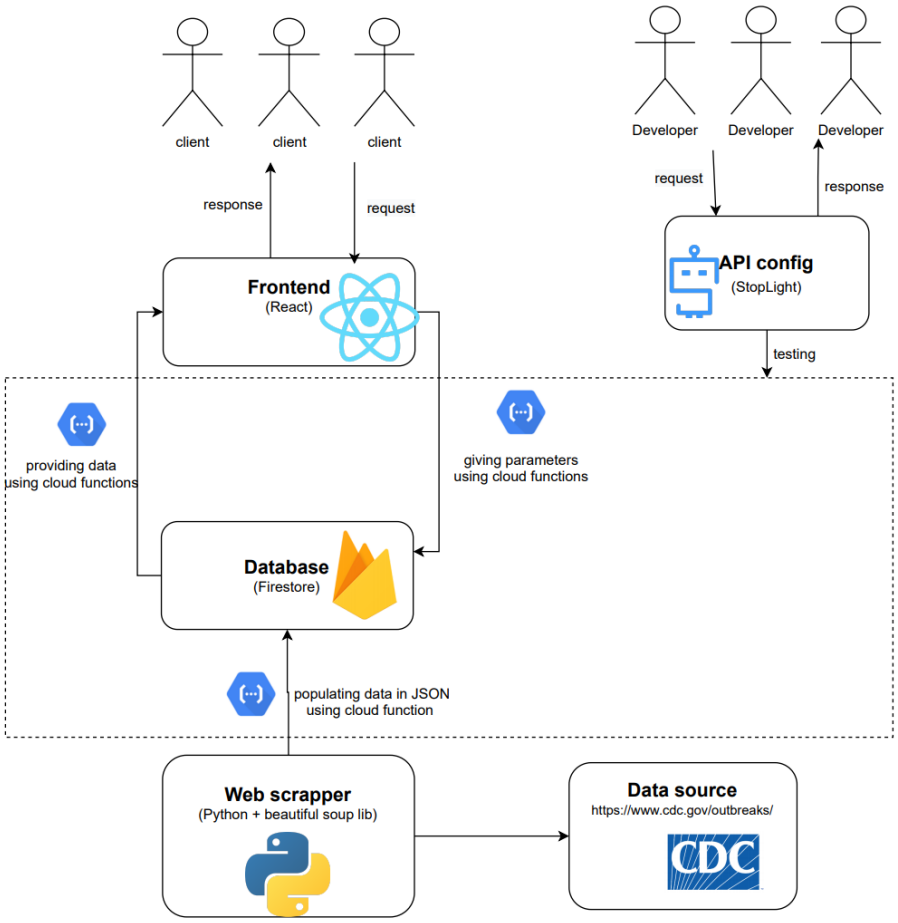
We will be updating the database using PUT cloud functions to make changes to objects that are already in the database. We will be updating it either on a regular basis (every 10 mins) or checking the last modified date to keep it up-to-date. We will also be building multiple GET methods with different 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

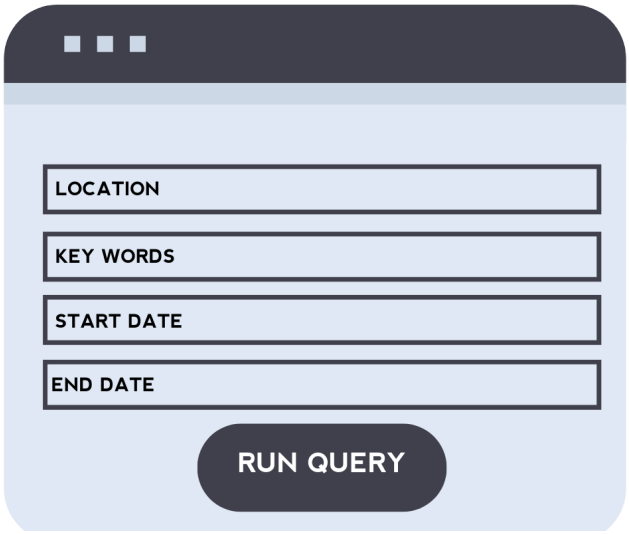

```
{
  "url": "https://www.who.int/csr/don/17-january-2020-novel-coronavirus-japan-ex-china/en/",
  "date_of_publication": "2020-01-17 xx:xx:xx",
  "headline": "Novel Coronavirus – Japan (ex-China)",
  "main_text": "On 15 January 2020, the Min
istry of Health, Labour and Welfare, Japan (MHLW) reported 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	Easy 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.

4. Final API Architecture

Overall API

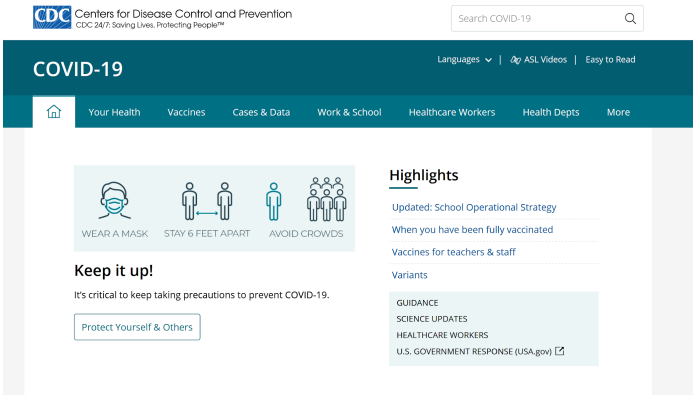
Our API was built to be easy and quick to develop, whilst still providing a user focused product. This is a product that provides the user the means of getting the right data, when they want it, how they want it.

Diagram of our final API structure:

Data Source - CDC Website

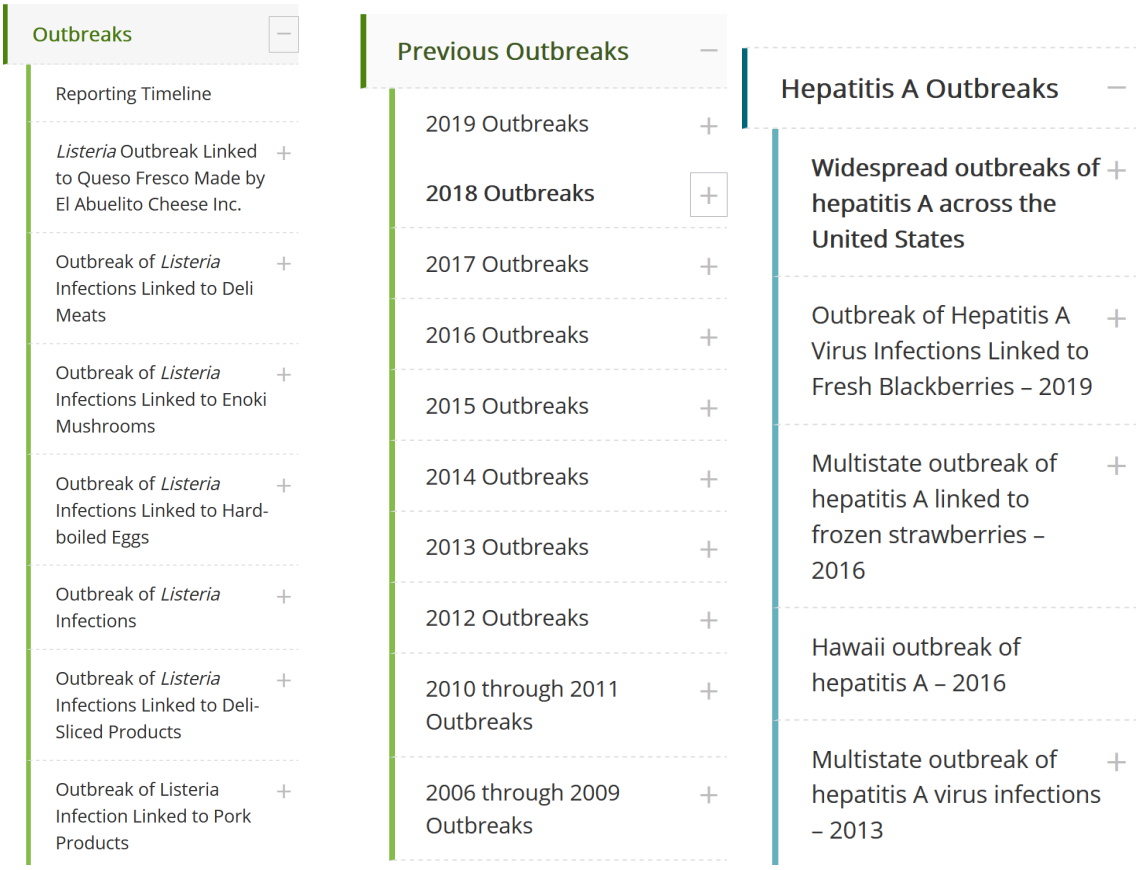
I. Limitations of this data source.

Our data source is the CDC. Since they do not have an API, our team uses a web scraper to retrieve the data. However this was a difficult data source to use because it has a few different page layouts (Shown below).



The above two images are examples of pages with vastly different layouts. As seen in the screenshot, the first page has a field for published_date shown on the page but the second one is somewhere hidden in the HTML. This means every time when we see pages with different layouts, we have to manually examine through their html and discuss the cases separately in order to extract the target information.

CDC webpages are not uniformly structured, and this situation happens more than we expected it to be. In general, information that has specific fields dedicated to includes ‘title’, ‘date_of_publication’, ‘url’ and ‘main_text’. They are relatively easy to extract. However, for information like “location”, “event_date”, “syndromes”, CDC does not have specific fields dedicated to them, therefore when scraping we will have to use other libraries or regex to gain results.



The above images is a comparison of the outbreaks section between Listeria, Salmonella and Hepatitis A. We can see that even though they have similar layouts, there are still minor differences in text. Using event_time as an example, if there are valid times shown in the titles, then we use them as the event_date. If not, then we have to examine through the main_text to find all of the phrases that look like a datetime, analyse their validity, then put one of them as the event_date.

Data Collection - Web Scraper

I. Implementation

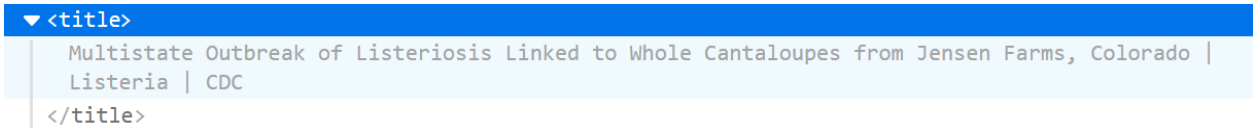
The Web scraper is implemented in python using request and beautiful soup as the main scraping tools. Despite their poor speed performance, they are easy to use when it comes to identifying tags and extracting text. For scraping CDC web pages in particular, the functionalities they provide in extracting target information from static HTML pages are more than enough.

The procedure can be summarized as:

1. Collect all links that we are interested, including the links on the base page as well as the nested links
2. Request for HTML page then scrape information from it.
3. Collect the information we scrape and put them into corresponding fields or objects.
4. Upload the data onto firestore using firebase credentials

.

During the process of extracting data, we imported various libraries to help us achieve the goal. For most of the pages that we scraped, the fields **title**, **main_text** and **publish date** are very easy to retrieve using beautiful soup and regex. Title is usually in between <title> tags, main_text can be scraped by searching for <p> tags, and publish date usually has a field called DC.date. Here is a quick example of how title is displayed in the HTML:



For the information that is dynamatic over different pages, we collected them by utilising a variety of available python libraries - They include: Fuzzywuzzy (compare the similarity between two words), Geograpy and Nltk (extract location names from text), Datefinder(extract datetime phrases from text). Geograpy and Nltk are used to extract location information about outbreaks, datefinder is used to determine the earliest event of the outbreak, and Fuzzywuzzy is used to compare the words in a given text with the identifiers provided in the spec to determine diseases. Here is an example of how disease is determined using Fuzzywuzzy:

Title: Outbreak of Listeria Infections Linked to Deli Meats
Identifier list: [{name: cholera}, {name: listeriosis}, {name: COVID-19}...]

We split the title by whitespace then loop through each word and compare with the name in the identifier list, Fuzzywuzzy library will be able to tell that the word “Listeria” is similar to “listeriosis”, therefore have listeriosis as the disease of the report.

However, there are flaws towards using these libraries. They are detailed in challenges and shortcomings.

Deployment is analysed at the end of scraper section.

Challenges:

The most challenging part of building a web scraper is to be able to cope with different page formats while ensuring the accuracy of the extracted information. In order to achieve this, we manually

compared pages with different layouts that our scraper might come across, and coded to ensure we covered as many cases as possible.
Another challenge was to come up with the best scraping strategy along with the imported libraries. For example, since CDC does not have a clear field for us to extract disease type, we needed to find a way to scrape the result as accurately as possible. The solution we came up with was to compare the similarities between words, which is highly effective but still has flaws.

Shortcomings

I. Scrapping outcome

Extracting location
To extract location we came across a Type I vs Type II error. We have tried two libraries Geotext and Geograpy. Geotext cannot do an in-depth matching. For example, it can't handle upper and lower cases and abbreviations. Due to these shortcomings of Geotext, we decided to use Geograpy. However Geograpy can be too in-depth sometimes. For example, it picks up the word "Ask" or "Standard" as cities. Thus, we had to manually exclude these cases, but similar cases may still occur, and lead to a result of having location with confusing names.

Event date matching
What we are doing now is first collecting datetimes from the title. If there is a result then we return the earliest time, if there is no result then we collect datetimes from the main_text. Again if there is a result then we return the earliest, if there is no result we then return the publish date. We are able to exclude invalid dates (e.g year being 800). However we can not ensure that the earliest date mentioned in the main text, is the starting date of the outbreak. We have made it so it is highly likely to be the case but it is not a perfect filter.

Disease matching
We compare strings of the title, word by word with the identifier list and return the word that has the highest similarity with the identifier list. It works most of the time but there are edge cases. We had to manually handle those cases and the issues mentioned above are mainly because webpages on the CDC are too dynamic and there are no specific fields for us to easily collect other than title, publish_date and main_text.

Long performance time

Since our scraper mainly uses beautiful soup and requests, along with the usage of many other libraries, it takes a long time to collect all the target data. It also affected our deployment - which is detailed in the deployment section.

II. How this differs from our initial design

In our initial design, we mentioned three points:

a. We were going to do a depth first search
b. We were going to scape everytime the aPI gets called
c. If no calls in a week it does it using cloud scheduler

For point a, during implementation, we realise that with the help of python beautiful soup library, we do not need to worry about the algorithm behind searching.

For point b, it is not reasonable since that discards the usage of firestore. It is also not realistically possible since the scraper takes a long time to run.

For point c, we are able but have not yet implemented due to the difficulties we are having with deployment. It is detailed in the Deployment section.

Deployment

Implementation

For web scraper, we initially had 3 ideas.

1 idea

- When user calls the API
- We call the web scraper endpoint
- Goes to the database to find the relevant URL's that need to be scrapped
- Compare the last scraped time with the last modified date of the HTTP request
- If page has been updated, then web scraper will scrape the page
- Updating the database

2 idea

- Google cloud scheduler
- Calls the function every 2 hours to make sure that we are up to date all the time

3 idea

- Every 2 hours via google cloud scheduler, the API is called we can run a checking endpoint to see if the website has been updated, if no then we do nothing
- If not we can go scrape it

During deployment, we decided to go with the second idea. This is because realistically we are not able to call the webscraper every time when users call API due to the long response time of web scraper, and it discards the idea of having a database. We are also short of time to implement the “last scraped” functionality. Additionally, CDC does not update their website very frequently, therefore 2h is more than enough to ensure the data in our database is updated.

Challenges:

The main challenge for now is that we cannot deploy it on google cloud due to the time limit they have. Google cloud functions only allow a 9 minutes runtime, however our web scraper takes around 20 minutes to finish. This is the biggest challenge for now, and we are still working on it. Some solutions include breaking scraper into different parts, using App etc.

Shortcomings

We only partially deployed the web scraper - it only worked with collecting “article” object but not “reports” or “locations” due to the runtime limit google cloud has. We are currently working on breaking web scraper into parts and deploying them separately.

Another shortcoming is that we do not have an AUTH key. This might result in security concerns.

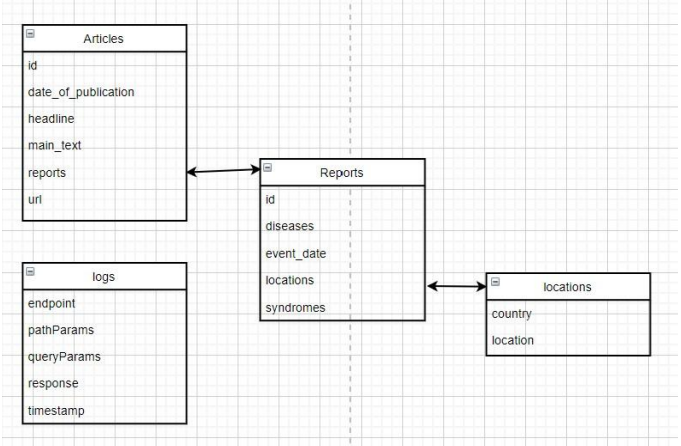
Furthermore other short comings include:

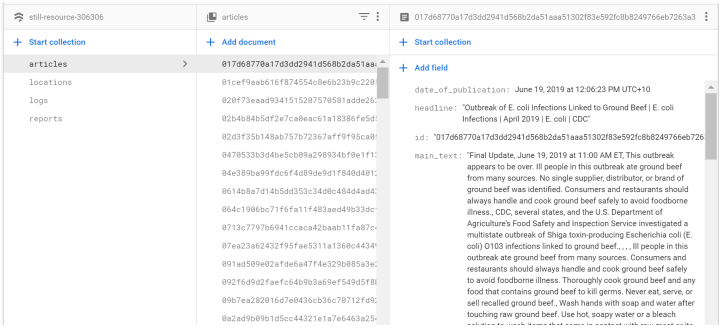
- Underestimating the time and knowledge it takes to

Data storage - Firestore

I. Implementation

The diagram on the left uses entity diagrams to describe how we are storing our data in Firestore.





We implemented our data storage through Google's Firestore. After we gather the data from https://www.cdc.gov/, we store it in this NoSQL Database. We stored our data in this type of database because it allows related data to be nested within a single data structure. This allows us when we are developing, to access the data very easily, easier than having a traditional fully relational database which requires complex table operations. In addition to this, firestore has a very easy to use user interface. Allowing us to easily understand how our data is stored and how to extract it.

Within this database we have four different collections of data: articles, locations, logs and reports. These are based on the endpoints we are going to implement. A general API search with period of interest, key search terms and location, we will just search through the articles collection to find the data. However if you are looking for a specific report or location we just quickly search our reports or location collection. Finally, the logs store the API search history. This means the API user can see what other people are searching. This method of storing data per endpoint, allows us to make the most of the NoSQL databases benefits, allowing us to easily return the right data to the user.

Challenges

For web scraping, challenges only occur at the beginning of the setup. The command line Google provided for installing firebase was not up to date therefore we had to manually find out which version to install, then upgrade. Other than that, firebase and web scraper interact very well.

Shortcomings

At the beginning we did not think about how regions or zones would affect our firestore. This lack of understanding made us host our Firestore database in the us-central zone. This means that the data is far away from our end users which are based in Australia. We are currently migrating over to Australia-southeast, however it is not yet complete. Since the majority of our users are based in the australia-southeast zone then hosting our web service here leads to lower latency and higher bandwidth to deliver content to our users faster.

Data service - Google Cloud Functions

To create a web service, we had to first think about what makes a web service good. Availability, ease of use, standardised input and output, using appropriate nouns to describe endpoints, and nested resources e.g. articles?id=1 and gracefully returned standard error codes are needed.

I. Overall Implementation

We chose to use Google Cloud Functions to host our web service. This did exactly what we expected, and allowed the data to be available all the time. In addition to this, since Google Cloud Functions, is inextricably linked to Firestore, this made development very easy. We were able to use the Firebase CLI to deploy our cloud functions, and the functions logger makes development very easy to debug.

We implemented the API web service using NodeJS. We had to switch from python3 because NodeJS was the only supported language in Firebase Functions. This mistake arose because we were under the impression that Google Cloud functions supported Python3 and Flask.

II. Advanced Querying

This feature set is applied to the /api/articles endpoint. The following allows the user to query the API in specific ways in order to get a more refined data set, specific to their needs.

Period of Interest

We had an interesting challenge with how we allow the user to input date strings. There are two different ways to input a date - a date range string or a singular wildcarded date string.

- **DATE RANGE:** when providing a date range, neither date (start or end) may contain wildcard characters (i.e. 'x'). Returned articles are within the start and end dates inclusive.
    - e.g. "2019-01-02 00:00:00 to 2020-01-02 13:00:00"

- **WILDCARDED DATE:** when providing a wildcarded date, the date must contain at least one wildcard character (i.e. 'x'), which represents any number from 0-9. Since wildcards can represent a range of values, a wildcarded date is in itself a query.
    - e.g. "2019-01-xx 12:xx:xx"

We allow the user to input data in these different ways, to give them more flexibility for their different use cases. For example, the wildcarded date allows a user to specify a year very quickly. They would only have to input:

"2019-xx-xx xx:xx:xx"

Rather than the lengthy:

"2019-01-01 00:00:00 to 2019-12-31 23:59:59".

Allowing our end user to develop and call our endpoints in a more effective manner, whilst still allowing them to have the precision of the date range.

Key Terms

Users can also filter by providing search terms, known as 'key_terms'. The user provides a list of comma separated strings to the API and will return all matching articles that contain ALL key terms. Key terms are also not case sensitive and trailing white spaces are removed before being used so that insignificant modifications of the word do not inhibit the user's usage of our API. In the future, we are looking to implement the ability to specify a key term OR key term, so a user may again have a more refined way of calling our API and receiving the right data for them.

An **above and beyond** feature we implemented is the ability to retrieve articles that exclude key terms as well. This feature is easily used by prefixing a '!' before the search term. Furthermore, these excluding key terms can be used in conjunction with normal search terms to provide the user the ability to retrieve finer search results.

For example, if key_terms="corona,!salmonella,flu,!sneezing" then this will search for all articles that satisfy all four criteria:

- **CONTAINS** "corona"
- **DOES NOT CONTAIN** "salmonella"
- **CONTAINS** "flu"
- **DOES NOT CONTAIN** "sneezing"

Location

Lastly, users can refine their search by specifying a location of the reports in the resulting articles. By providing a string, this will return any articles that contain reports containing the provided location. This is also not case sensitive and removes trailing white spaces to ensure a more efficient user experience.

III. Shortcomings

There are several shortcomings to do with our API currently. Right now, we do not allow for pagination from requests, we only slightly filter before getting the resource from firestore, various security issues and do not have caching no versioning implemented. However, our web service meets our current sprints goals: to provide the user a functioning API that they can access at all times, and query using various endpoints and parameters.

Pagination

In terms of efficiency, we do not have pagination, which entails that the user could request a smaller amount of search results leading to a faster response time.

Filtering before querying firestore

After receiving the API request, we could use this request exactly to query the Firestore database more effectively. Currently we have a filter that only retrieves entries from the date range specified. For example if the path parameter period_of_interest was:

“2019-xx-xx xx:xx:xx”

The request to Firestore would be any articles within this date range:

“2019-01-01 00:00:00 to 2019-12-31 23:59:59”.

We could filter the resource more before calling the firestore base. We could filter it by location and key terms. This means the processing will be faster because NodeJS does not have to receive the barely filtered database every query.

Caching

Next is caching, which would bring the data closer to the user again, if someone else nearby or that user has requested that data before, it will not have to get the data from the server again, but rather, the data is located in a much closer location. Again, increasing the response speed. This is a simple fix about adding a few headers, so will be completed in the next sprint.

Security

In terms of security, we have chosen a very loose cors policy, currently accepting all other websites. This is because we have not implemented OAuth or other authorisation, and our priority is to make the data as accessible as possible. In addition to this, we do not have any POST, PUT or DELETE requests, so an attacker should not be able to alter the database, however we could be attacked with a denial of service, forcing us to go above the free tier of Cloud Functions and have to pay for these requests. A way of stopping this is to have users login and rate limit them. This means if a user (that is logged in) queries our endpoint 100 times in a minute, it is probably an attack and not a user actually trying to get the data. Instead of letting them complete these requests, we can not allow this user to send any requests for a time limit that grows exponentially. For example, the first limit can be 1 minute, then 2 then 4 and so on.

Versioning

Finally, versioning is key to allowing the user to adapt to change when they are ready and allow us to test versions of code before releasing it to the user. However, this does not matter as of yet, because we have zero users, but very soon we will implement this.

Errors and outages

In terms of error handling and outages, Google cloud also allows us to host our service in multiple regions so that in the case that the servers in one place go down we can handle the outage

Future Implementations

In the future, the above shortcomings will become our backlog of tasks to do.

IV. Challenges

The API was harder to implement because we are not as familiar with NodeJS and Javascript as we are with python3.

API Response

I. Implementation

We are building a web service for a group of people that are united by similar API responses. Therefore we should also build our API as close as we can to this design. This would make it easier for our users to integrate our product into their different use cases. This is due to the fact that they are already familiar with the response.

For example the following search returns:

GET: https://us-central1-still-resource-306306.cloudfunctions.net/api/articles?period_of_interest=2019-01-11 01:32:xx


```
{
  "log": {
    "team": "SENG3011_LINKED_LIST",
    "time_accessed": "2021-03-26T04:15:15.892Z"
  },
  "articles": [
    {
      "id": "558f2efbbd7501579437a3fe8bab5e3f958d7e4f113ed389681f8f4cae21e4cf",
      "date_of_publication": "2019-01-11T01:32:15.000Z",
      "url": "https://www.cdc.gov/ecoli/2008/ground-beef-kroger-7-18-2008.html",
      "main_text": "NOTICE: This outbreak is over. The information on this page has been archived for historical purposes only an
      "headline": "July 18, 2008: Investigation of Multistate Outbreak of E. coli O157:H7 Infections | E. coli CDC",
      "reports": [
        {
          "locations": [
            {
              "location": "New York",
              "country": "United States",
              "id": "9b5a85e97e956a1e337873c89d65f495cca8a70db6ca8f9c8da34737f64e3954"
            },
            {
              "id": "7104b63491def7a43494dae9ce7db2c0db0c0f63b8cac1ca06a6d5edefe85ee4",
              "location": "Indiana",
              "country": "United States"
            },
            {
              "id": "932005b7d7b235a1cb13a5edd139ce17c20f73ea4fd2ce24b62e523c59e29532",
              "location": "Ohio",
              "country": "United States"
            },
            {
              "id": "a0973d48f951d159958a59dcdff8fe817df6ff1c2443f224d399b076f68fac6a",
              "country": "United States",
              "location": "Michigan"
            },
            {
              "country": "Georgia",
              "id": "29758482b1b7b53689d5ff399979db011e2fc2260f89fb0f69e5e3e7b31d0f9a",
              "location": "unknown"
            },
            {
              "id": "378beee5676359235d37951c5e9d34ee1fd0fa7816088e7b602b033d45ed4947",
              "location": "unknown",
              "country": "United Kingdom"
            },
            {
              "location": "unknown",
              "country": "Brazil",
              "id": "60fbe59ef71c77cabba3e1a72c61334b529f8aa8f260c371eeb76f08be464b9d"
            },
            {
              "location": "unknown",
              "id": "702ad8c11230055534be05425cb45231d5f3be9b751e255fbb13c0b13ec9545e1",
              "country": "Jamaica"
            }
          ],
          "event_date": "2008-07-18T00:00:00.000Z",
          "syndromes": [
            "unknown"
          ],
          "diseases": [
            "ehec (e.coli)"
          ]
        }
      ]
    }
  ]
}
```