# SENG3011 Deliverable 2 - Final Design Report

Group: FatDonkey

Members: Flynn Zhang, Edwin Tang, Frank Su, Josh Rozario

## About Us

Due to unprecedented events from the past year, our team has been tasked by ISER (Integrated Systems for Epidemic Response) to develop an outbreak surveillance system to further assist their Epiwatch software which uses public data sources to detect outbreaks.

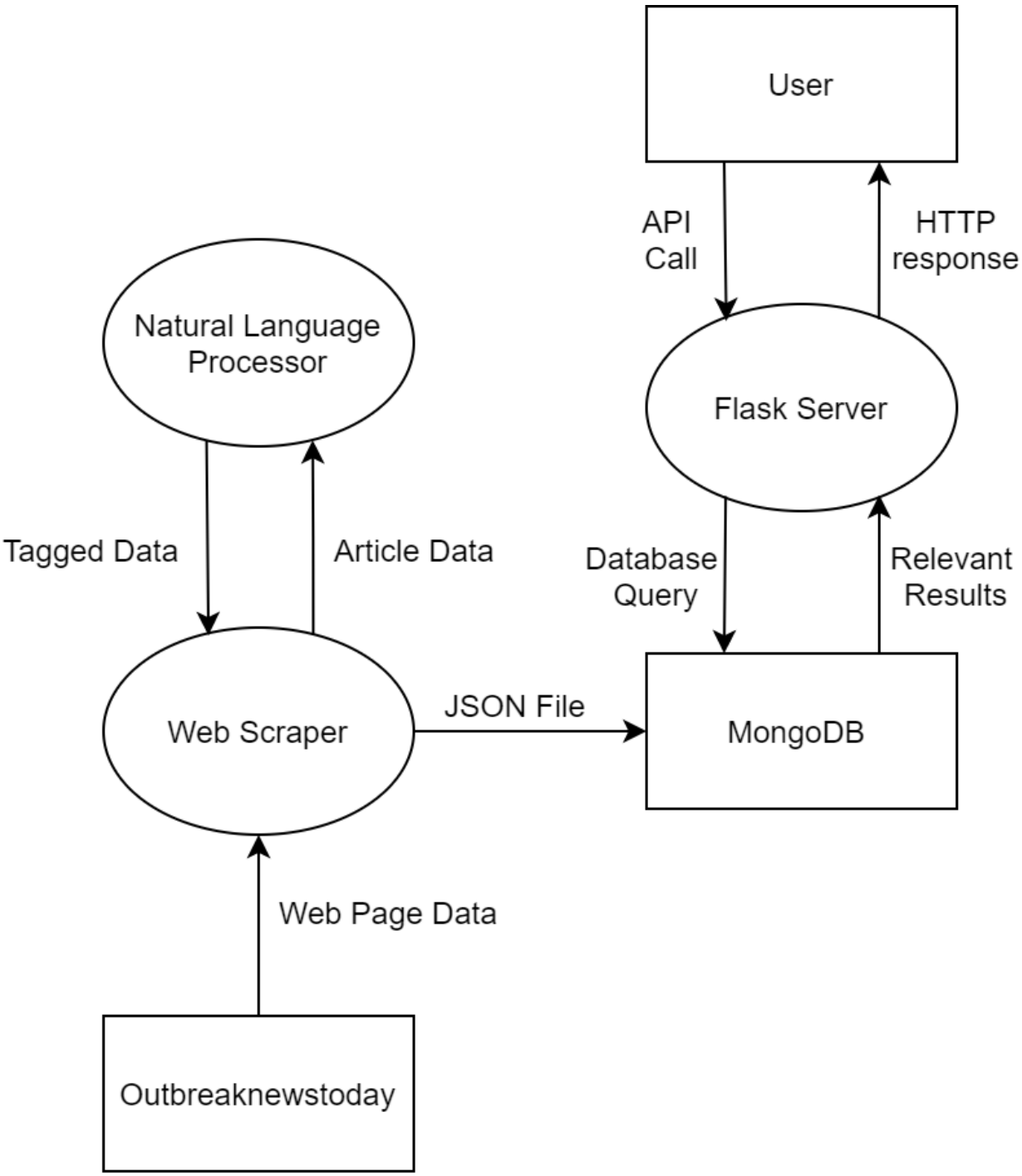
The following report details stage 1 of our project which is to develop an API which retrieves disease reports from a data source (outbreaknewstoday.com) that are relevant to the parameters parsed to it. Our rationale for programming language, hosting, storage and framework decisions will be included.

In stage 2 of our project, we will be developing a web application which allows users to interact with our API in a user friendly way.

## System Overview

Initially, the articles from the site will be recursively scraped and stored. After, all articles will be passed to a natural language processor to extract  the name of the disease and the location of the disease and add that information to each article element. The tagged data will then be stored in a database for easy retrieval from our API. A user will be able to call our web server after it has been deployed to request their desired data. In order to maintain an updated database, we will scrape and tag articles once a day.

The following diagram is a visual depiction of how our system will operate:


User
API
Call
HTTP
response
Natural Language
Processor
Flask Server
Tagged Data
Article Data
Database
Query
Relevant
Results
Web Scraper
JSON File
MongoDB
Web Page Data
Outbreaknewstoday

# API Module

We used Python and Flask to develop our Restful API and scraper as we have all used the framework and language in the past.

Scraping the data

The scraping of our data source http://outbreaknewstoday.com/ will be carried out using Scrapy, a popular Python module that our team also has experience with. It will extract the date,url, region and title of the article as well as the body of news reports. Extracting the metadata will help categorise the reports and help users of our service refine their search. We used CSS selectors to extract information from the HTML elements which contained our desired data. E.g. **response.css('div.posttitle').css('h1::text').get(),** The website had around 5-6 links to related articles at the bottom of the page so our crawler would open and scrape each related article. Then the scraper would repeat the process and scrape the links at the bottom of each of the previously mentioned pages. This process would repeat many times. Due to the recursive nature of this scraping, our scraper was able to scrape around 8000 articles in a few minutes, some articles scraped were written as early as 5 years ago. The scraper would store all the scraped results as a json file.

Cleaning/tagging the data

We believed there would be certain challenges in extracting the location and disease name metadata as this information is embedded within the title of the news report.
To solve this problem, we will use text analysis AI (Spacy) to extract the desired data. Spacy is a python library that conducts natural language processing (NLP). We had originally planned to use another library called NLTK but it's accuracy in extracting our desired data was not up to our standards. We needed Spacy to extract 2 critical pieces of metadata which were the **location** of the reported outbreak and the **name of the disease** that was reported. These 2 tasks fell under a subcategory of NLP called **Named Entity Recognition**, essentially the identification of proper nouns. Named Entity recognition works by using a model supplied by the user to quickly label words in a sentence.
For example,


In this sentence, the model was able to correctly tag the word "Apple" with ORG meaning that it recognised Apple as the name of tech company and "UK" as a geographical location. (GPE tag).

We used the default **en_core_web_sm** model to identify the names of countries or states. Our tagger would look for the first word in the title of the article with the “GPE” tag and set that as the location of the outbreak. If it was not able to detect a location in the title, it would try the first sentence of the body of the article.

Identifying the location of the outbreak was the easier of the two tasks as it is a very common feature that programmers might use. As a result, the accuracy of the tagging was quite good. Identifying the name of the disease was much harder as it is quite a niche area in named entity recognition. However, we were able to find a Python package called **scispacy** which had models specifically related to identifying scientific words. In particular there was a model called **en_ner_bc5cdr_md** which was built for identifying diseases and chemicals. We used this model to identify the name of diseases and it still had some trouble identifying diseases so we applied the earlier strategy of looking for a disease in the title first, then the first line of the body if it was not able to find it in the title and then in the second line of the body as a last resort.

With the use of spacy and scispacy we were able to tag 70% of articles with a location and disease with about 2500 unique articles not having either a location or a disease tag. 700 articles did not have a location nor a disease tag.

The database

We imported the tagged data json file into MongoDB, our NoSQL database management system which was perfect for processing our document based data (json files). Using a DBMS allowed the API to efficiently query this database without having to access each individual element inside a json file whenever a call was executed. From there onwards the python driver Pymongo provided by MongoDB allowed us to modify the queries to the database similar to how one could modify a SQL query to retrieve specific results. This served as a way to process and retrieve the relevant articles in the database as per the user inputs passed to our API endpoint.

Webserver

Our API was built using Flask, a lightweight web server framework. When a call is made to our flask server, the arguments of the query string (if any) are parsed and checked for their validity. If any of the inputs are invalid, the server would respond with a 400 error code and an appropriate error message. If the inputs are all valid, the server would search the database based on the given parameters and return the results .

We built individual functions for each parameter that could be given to the server. For example, there are separate functions for filtering based on location, disease name, etc. The main function would just call any combination of these functions to obtain the desired data. This allowed concurrent development of the functions and introduced modularity to the design of our server. This proved highly useful during the implementation of our API as it meant that multiple called functions serving different roles that aided the main function could

be worked on at the same time, which drastically decreased production time and led to our timely implementation of our system. The modularity of our implementation also allowed us to test each function in isolation as to remove any unnecessary variables during testing that could cause errors. As such we were better able to isolate any errors allowing us to fix them more easily. If the request was successful, the response from the server would be a list of articles in JSON format.

We used the Flask-Restplus extension to automatically document our API endpoint through a Swagger UI. However, many aspects of documentation needed to be manually added. The response codes and messages were added through decorators.

```
@api.response(200, 'Success', article)

    @api.response(400, 'Bad request', error_msg)
    @api.response(404, 'URL not found')
    @api.response(500, 'Internal Server Error')
```

Similarly, the description of the  input parameters was also added through decorators.

```
@api.doc(params={'location' :'Country or
State/Province (e.g. China)',
                 'disease' : "Type of Disease (e.g.
Cholera)",
                 'start date': 'Outbreak reported on
or after this date (dd/mm/yyyy)',
                 'end date': 'Outbreak reported on
or before this date (dd/mm/yyyy)',
                 'region': 'Geographic region of
outbreak (e.g. Europe)',
                 'results': 'Number of results (Must
be > 0)(e.g. 10)' })
```

Flask-restplus allowed us to add “models” to describe the structure of the response of our API. We were able to model the structure of an individual “article” object and display it on the Swagger UI.

```
article = api.model('article', {
    "title": fields.String(example="SARS, MERS ruled out in China pneumonia cluster"),
    "date": fields.String(example="January 5, 2020"),
    "location": fields.String(example="China"),
    "region": fields.String(example="Asia"),
    "url":
fields.String(example="http://outbreaknewstoday.com/sars-mers-ruled-out-in-china-pneumo
nia-cluster-40965/"),
    "disease": fields.String(example="SARS"),
    "body": fields.List(fields.String(example="Health officials in the city of Wuhan in Hubei
province, China ")),
  } )
```

We were also able to log the requests and responses of the server and write it to two files using decorators which would trigger functions before and after a request. The logs are captured in two files, one that is simpler, easier to read, the other more verbose and contains more detail. Details such as how long the request took to process are recorded.

Deployment to the cloud

We deployed our API to the web using Heroku's Cloud application platform for its convenience and widespread availability. Deployment using Heroku, required a requirements.txt file which had a list of dependencies that Heroku needed to install for our application to work. Our app was successfully deployed using the Heroku-20 stack for Python applications and can be found at this link: https://disease-watcher.herokuapp.com/

## Module Interaction

We developed our API as a RESTful API as it is the de facto standard for creating the architecture of network systems. Our API only has 1 endpoint which is the GET method the user will use to pass certain parameters for the articles that they desire.

Our API takes the following parameters:

'location' :'Country or State/Province (e.g. China)',
'disease' : "Type of Disease (e.g. Cholera)",
'start date': 'Outbreak reported on or after this date (dd/mm/yyyy)',
'end date': 'Outbreak reported on or before this date (dd/mm/yyyy)',
'region': 'Geographic region of outbreak (e.g. Europe)',
'start_index': 'Articles start from this index (Default is 0 which is the first article)',
'end_index': 'Articles end at this index (Default is len(articles) - 1 which is the last article)'

In order to prevent significant latency when using our API, the user is not able to request all articles in the database and can only obtain a subset of articles by supplying the “location”, “region” or “disease parameter. All other fields are optional. The **start_index** and **end_index** parameters allows the user to see a portion of the total results that would be returned if no indices were set. For example, if the user wanted to see the first 5 results they would set the start_index to be 0 and end_index to be 4.

Valid **region** inputs are: 'Africa', 'Asia', 'Australia', 'Canada', 'Europe', 'Indian subcontinent', 'Latin America and the Caribbean', 'Middle East', 'US News'

Our module will find the relevant articles that matches the user’s parameters from the database. The API call will then be processed and return to the user, a list of the relevant articles as a JSON object. JSON objects are used by most servers and clients to exchange information, thus allowing our API to be more compatible with more users and increasing our user market.

If there is an error during the execution of the API request, it will be handled as per HTTP response codes. Our API has the following error codes:

**200** - Success

**400** - Bad request (inputs are invalid or missing)

**404** - URL not found (invalid route)

**500** - Internal Server Error

In particular, a 400 response code would be accompanied with an error message, explaining which input was incorrect.

User Interface

A user can easily test our API using the Swagger UI that has been provided at our API URL, by typing in the parameters and clicking the “Execute” button. The UI is also able to provide the URI path and query parameters for developers if they wanted to execute the call in their code.
E.g.

https://disease-watcher.herokuapp.com/outbreak/?location=China&start%20date=20%2F04%2F2019

**API Call Examples**

curl -X GET  "https://disease-watcher.herokuapp.com/outbreak/?location=China" -H "accept: application/json"

Accept: text/html
Accept=Language: en-us, en

curl -X GET
“https://disease-watcher.herokuapp.com/outbreak/?location=China&start%20date=20%2F04%2F2019" -H "accept: application/json"

Accept: text/html
Accept=Language: en-us, en

**JSON Return Example**


[

  {

    "title": "Gates Foundation commits $10 million to contain the global spread of 2019-nCoV",

    "date": "January 27, 2020",

"location": "China",

"region": "US News",

"url": "http://outbreaknewstoday.com/gates-foundation-commits-10-million-to-contain-the-global-spread-of-2019-ncov-42552/",

"disease": "unknown",

"body": [

"The ",

" announced that it is immediately committing $10 million in emergency funds and corresponding technical support to help frontline responders in China and Africa accelerate their efforts to contain the global spread of 2019-nCoV.",

"The foundation is committing $5 million to the 2019-nCoV response in China and is already working with a range of Chinese public and private sector partners to accelerate national and international cooperation in areas of critical need, including efforts to identify and confirm cases, safely isolate and care for patients and accelerate the development of treatments and vaccines.",

"Partners include the National Health Commission and Chinese Center for Disease Control and Prevention, the National Natural Science Foundation of China and various research institutes affiliated with the Chinese Academy of Sciences, Xiamen University and Sinopharm China National Biotec Group.",

"The foundation is also immediately committing $5 million to assist the Africa Centers for Disease Control and Prevention in scaling up public health measures against 2019-nCoV among African Union member states. These measures will include technical support to implement the screening and treatment of suspected cases, laboratory confirmation of 2019-nCoV diagnoses and the safe isolation and care of identified cases."

]

},]

## Implementation Decisions and Rationale

a. Python
    i. All group members have familiarity with Python and so we can minimise the amount of time spent learning the language
    ii. Language is suitable for the scope and agile development of our project
    iii. Existing libraries will streamline development of our program as various functionalities (e.g. Natural language processing) can be easily provided by Python libraries
    iv. De-facto programming language to interact with the web.
b. Flask
    i. All group members have familiarity with this library from previous projects.
    ii. Serves as easy way to process API calls through implementing them as HTTP endpoints
c. Flask-restplus
    i. Python library to automatically document our API as we develop it by providing a Swagger UI.
    ii. Faster than writing out all of the Swagger UI documentation.
d. MongoDB
    i. Simple and effective NoSQL database management system but is still powerful that will allow us to easily store the scraped data.
    ii. Can be easily set up to interact with Python through Python scripts
    iii. NoSQL can provide superior performance in comparison to relation DBMS
    iv. Does not require data to be transformed into a schema set by the database
    v. Existing data stored in JSON file can easily be imported into a MongoDB database
e. PyMongo
    i. Python library to allow interactions with the MongoDB database
f. CSE Vlab
    i. Default development environment for this project as many of the tools going to be used are already available
    ii. Using CSE Vlab ensures consistency in the development environment among all group members
g. Scrapy.py
    i. Python library for web scraping that will allow us to scrape data from our allocated data source

    ii. Familiarity with this library from previous projects will streamline the development of our scraping program
h. Heroku
    i. Our application is planned to be deployed on Heroku as it is more suitable when considering our project scope in comparison to AWS.
    ii. Allows for easy and flexible management of our system
    iii. Uses Ubuntu 20.04 (default version) as the operating system for the web server which will be supported in the long-term (2025)
i. Spacy
    i. Python libraries to help us easily extract the disease names and locations from the title of each data source
    ii. Can experiment with a variety of prediction models to find one that is the best at identifying desired metadata
j. Regex
    i. Regex used to match strings of user input to the articles stored in database

# Challenges

Classifying articles/Data processing

Extracting the metadata from articles is an important task as we believe having a single field to identify the location and disease mentioned in the article is critical for fast database queries.  Performing this task has been quite challenging as the models used are not very good identifying disease names as diseases may be abbreviated (e.g. H9N2) or may have different forms (e.g. Coronavirus vs Covid-19).

We have had more success with identifying locations as the models are more familiar with geographical names but there are also difficulties involved. Locations that are identified can be cities, states, countries or even continents which creates inconsistencies in the type of value that the field may hold. This could result in queries missing certain articles that could be relevant to the search.

Another issue related to processing the raw data from the site, is that the body of the article may mention more than 1 location or disease. Adding these values to the metadata of an article would slow down the database query as the database would need to carry out multiple comparison operations in the event of a query. The complexity of this kind of search would be O(**m** x **n**) where **m** is the average number of tags per article and **n** is the number of articles. In comparison, if an article only had 1 value per article the complexity would be O(n). In the end we settled for only having 1 report per article to maintain the speed of our search.

High Coupling

The API is over reliant on the functioning of the scraper and its consequent text analyser to perform its role accurately in order to function. Furthermore, the flask server is reliant on the correct functioning of our database so it can accurately return the correct information to the user. Evidently, each component of our system is reliant on another and a malfunction of a single component may have a domino effect on the rest of the system and will affect the endpoint.

This problem is difficult to address due to the nature of the system. Since, our system only consists of 3-4 components (web server, database, natural language processor), we believe this is an acceptable design flaw since we have the necessary manpower and time to ensure that each component performs as intended. This problem may become more pronounced if our system is scaled up to include more components in the future.

Deployment

When we deployed our API to the web, Heroku needed a requirements.txt file which was a list of dependencies(python modules and their versions) so that it could download all the modules that were used by our API. However, we did not have this file because we were not using a virtual environment during development to isolate our dependencies for the project. The only solution was to run a script that created requirements.txt file with all Python modules that one of our team members had ever downloaded. As a result, there were many modules that our API did not use and was not required for Heroku to download. We had to manually filter all the dependencies that were relevant to our project. We have learnt from this experience and will definitely use a virtual environment to keep track of our dependencies for Phase 2.

Development of API

There were several problems we faced while creating the webserver. Our biggest hurdle was the implementation of our code in general in which our spec required us to create an endpoint where users can search up articles through various fields such as date of article, location of article, disease etc.. We as a group also decided to allow users to leave some fields blank when searching up articles. This proved quite difficult to code, leading to initial difficulties during development and subsequent poorly written code. Following our first iteration, we encountered several issues in the code which made implementing extra features difficult. However in following iterations a lot of the code was cleaned up and refactored and many of the difficulties in implementing new features are no longer present.

# Shortcomings

Search algorithm

Another shortcoming to our webserver is the inability to search up terms regardless of case. This was mostly due to the limitations of the mongodb collection functions we used. We discussed the alternatives of using java list functions or simply iteration via a loop through our mongodb database instead however this solution would have drastically increased the time of our queries as we would have to loop through the entire database several times when a query is made. We also discussed using certain workarounds found using regex however we quickly realised that this was very dangerous as it seemed as if it was potentially possible for our regex solution to pick up terms and articles that we did not actually want to pick up. Hence, we settled that our user must type in the exact parameter they want to match, matching our database's cases exactly.

In order to match dates however, we were forced to use the described looping solution as our scraper stores the date of an article in the database as a string. As a result, when we were to match articles to dates we had to iterate through each article in the database, manually convert them to a datetime format to then compare them to the datetimes the user has given to us. One fear of this method was the drastically increased runtime of the query. However as this looping method was only used in one function, our queries were not noticeably long.

Classification of articles

Since our Named Entity Recognition methods were not able to correctly identify the location or disease for every article in our database, a significant portion of our database lacks either both tags or one of the tags (around 3500). Therefore, we have many articles that will not get picked up in a query and are simply taking up space in our database. However, this is not without reason as every article has a **region** tag which was scraped from the article metadata. The region tag was very inconsistent in the type of values it would have. For example, it's value could be a continent (Europe), or a country (Australia) or a subcontinent (Indian subcontinent) or something random like (Medical History). However, it would provide a baseline parameter that every article would have and which a user could use to search very broadly.

We were also not able to obtain more than one reported outbreak per article as we believed that it would be more efficient performance-wise to only have 1 reported outbreak per article (as discussed in the Challenges section).

Updating the database

Due to time constraints on our project, we were not able to implement automatic scraping, tagging and importing of articles into our database so that our API could be as up to date as

the website http://outbreaknewstoday.com/. We had an idea of how we could write scripts to automatically do the data processing and importing but did not know how to schedule this to happen everyday through Heroku. We decided on finishing the primary functionality of our API and writing documentation instead of developing this feature.