DESIGN DETAILS

SENG3011 REPORT

TELEUBBIES
Yin Huey Tan z5211373
Amanda (Xiaorui) Li z5206613
Sarah Oakman z5206178
Lavanya Sood z5208121
Yiyun Yang z5187469

INTRODUCTION

The **EpiWATCH system** utilises multiple accessible data sources to automate epidemic detection. We are creating a REST API module to contribute to the **EpiWATCH system** by identifying disease reports from the WHO data source and in the future, developing a web platform to bring epidemic detection to end users.

The design phase is necessary in projects to discuss how the whole system will be developed and deployed. Python, Flask, SQL and JavaScript will be used as the development languages and various tools including Swagger and Postman were considered for the API documentation and testing. We also explored ways it could be run in web service mode and all types of parameters that can be passed into the API module and how the results are collected.

Changes to our architecture since the last deliverable are bolded in each section.

1.0 API DEVELOPMENT AND IMPLEMENTATION

1.1 WEB SCRAPING

The first step for developing an API module is to translate information on a webpage into data we can search and sort through. In order to 'read' and interpret the data and articles on WHO which will later be stored in databases, we need to create a spider (web-crawler) through Scrapy which is able to view and save text on website pages, as well as traverse pages. Scrapy crawler works by making a request to the URL (https://www.who.int/csr/don/archive/year/en/), traversing through all the years using Beautiful Soup and looks for specific information in the articles (e.g. report content, dates, key terms etc) within the HTML file through the CSS and XPath Selector (e.g. based off what the div classes are called). It then creates Scrapy Items which resemble Python dictionaries with the extracted information and continues the process through calling another request to the next article found by Beautiful Soup.

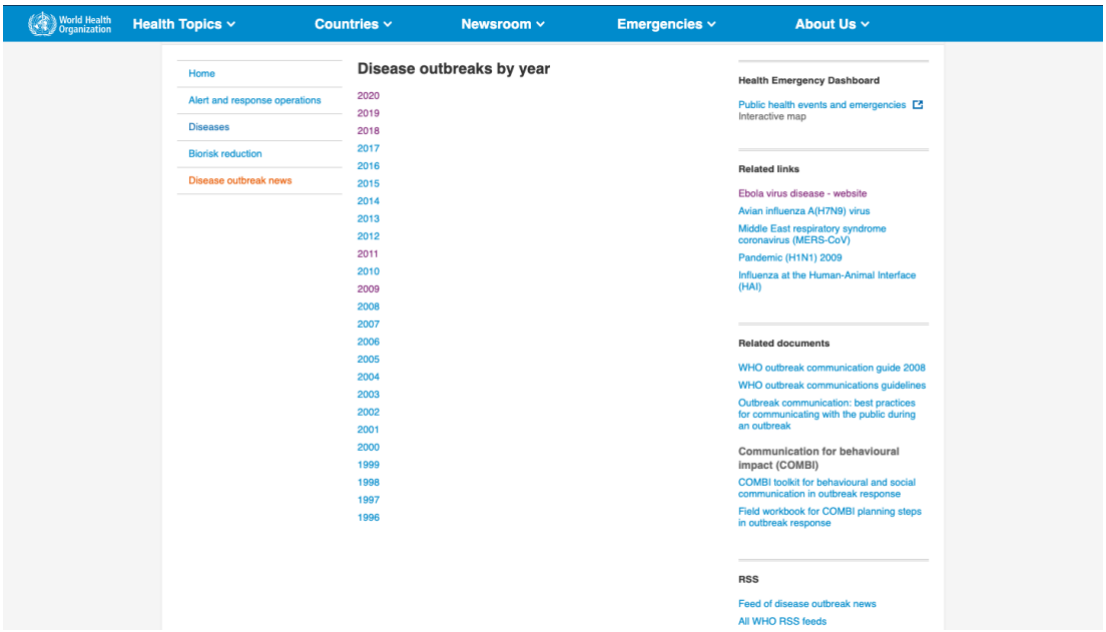

Fig 1.0 – Starting page separated by years which links to all other articles

Once the information is saved in Scrapy Items, it can be extracted and cleaned/sorted using Item Pipelines. It also allows data to be checked for duplicates and validates if it contains the right fields. It is then inserted into the database using SQLite3 in Item Pipelines.

We split these tasks between three programs (reportbot, linkbot and updatebot). Reportbot looks at an article, scrapes for necessary data e.g. headline, main text separated by disease reports within article and writes to the database. Linkbot runs through all links on WHO, calling reportbot to scrape each and every article. Updatebot looks for new articles published after the last scraped time which aren't in the database and calls reportbot for these links. This process was streamlined since WHO sorts articles in order of most recent article first.

Updates to the scraping architecture from the last deliverable include removing the use of feed export (no ouput JSON file needed, all done with Scrapy Items) and using Beautiful Soup to traverse links and update database regularly.

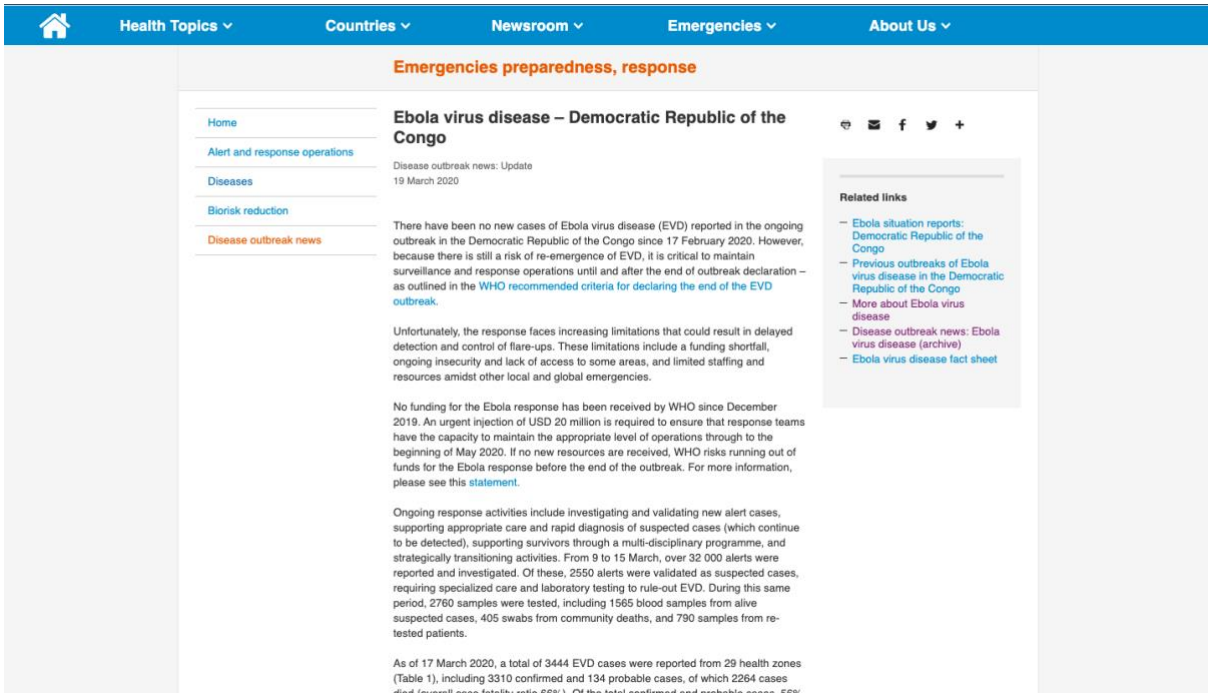

Fig 2.0 – Article containing information to be scraped

1.2 DATABASE

Another development aspect that needs to be considered for the API module is the database. The database structure is described below with all necessary fields outlined in the project specification and the WHO criteria included. The diagram outlines the relationships and keys between the tables and the fields present. Also, fields such as the PublicationDate, Location, Disease and Search Term will be indexed to allow for more efficient searching when parameters are given.

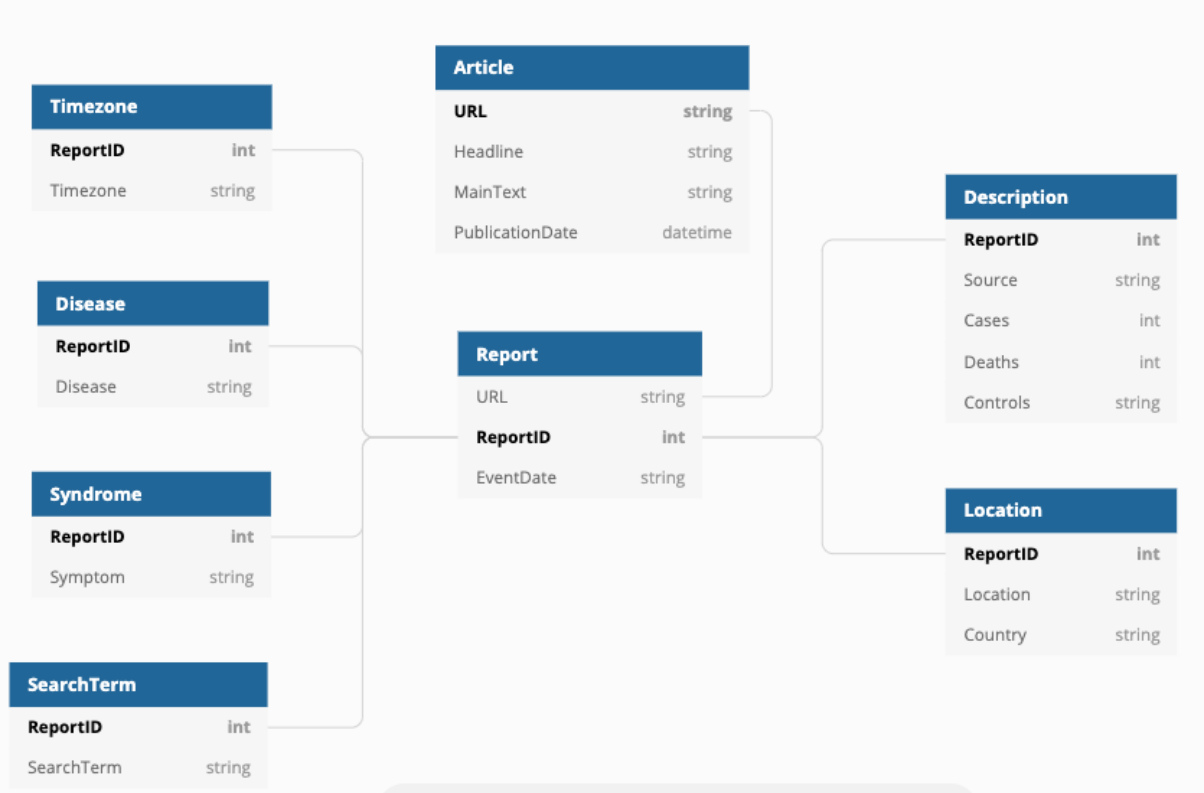

Fig 3.0 – The design and structure of the database

SQLite and SQLite Studio will be used to store, manage and show the information within the database. SQLite was selected since team members are familiar with the program, it can easily be imported and used within Python and it is compatible with several operating systems including Mac OS and Windows which is what the team will be using. SQLite Studio will be used to help visualise the database since it is built to work with SQLite, and it is also compatible with the team's operating systems.

Updates to the database since the last deliverable include adding a Timezone table, changing EventDate field from Datetime to String to account for inputs concerning a range of times, and adding a Country field to Location to follow the specifications.

1.3 API DEVELOPMENT

To develop and build our API, we will be using Python with Flask since everyone in our team is familiar with this language and framework. We will be using Flask to map our HTTP requests to Python functions. The development of our API will be following the REST principle which is based on the four methods defined by the HTTP protocol : POST,GET,PUT and DELETE which correspond to the four traditional actions performed on data in a database: CREATE, READ, UPDATE, and DELETE.

In order to connect our API to our database, we will be importing sqlite3 library in python. An object representing the connection to the database is bound to the `conn` variable. We then create a cursor object which moves through the database to pull our data. Finally, we execute an SQL query with the `cur.execute` method to pull out all available data (`*`) from the table of our database. At the end of our function, the data is jsonified and returned in JSON format.

All information remains the same as previous deliverable.

1.4 API TESTING

We will be using platforms like postman, swagger and pytest to test and validate our API request-responses at the development stage. This allows us to double check if our endpoints work as intended in different scenarios during the development process.

Testing for scraping was done manually through collating a diverse range of reports with different formats as test cases. Once these worked, Python scripts were written to check linkbot went through all links and called reportbot correctly. Updatebot was verified by checking first it worked when ran more frequently (after 10 minutes), before running it daily and checking database was correctly updated.

Updates to testing since last deliverable include creating Pytests as well for error checking which uses SQLite3 to grab information from the database and expanding on manually unit testing for web scraping.

1.5 WEB SERVICE MODE AND API DOCUMENTATION

To run the API module in web service mode, Heroku and the Swagger User Interface will be utilised. Heroku is a suitable cloud-based platform for delivering the API in web service mode as it can be linked to GitHub, supports Python and frameworks that are going to be used and it's free plan is suitable for our use.

Heroku will be used to deploy the API module to a server allowing it to be accessed on the Web. This will be achieved by linking the team's GitHub repository to the deployment method on Heroku and also within the API module. After providing a project name, Heroku supplies a URL which can be used to run the API in Web service mode.

The Swagger User Interface will then be used to document and show all of the requests that a user can make using the API module, such as a request to get all disease reports in a given date range. Since the API will be programmed in Python with Flask, to make it available on Swagger the 'flask_swagger_ui' library will be used, specifically the 'get_swaggerui_blueprint' module. To provide the API information and details with Swagger, a json document will be referenced by a static path entailing the API requests available. Once the Swagger blueprint is registered, it can be accessed and used to run and test the API module. This would provide users with proficient information to use the API module for disease outbreaks.

All information remains the same as previous deliverable.

2.0 INTERACTIONS WITH THE API MODULE

2.1 API MODULE PARAMETERS

Parameters are passed to the module with the HTTP endpoints which will affect the API response by finding and presenting suitable disease reports and information. The parameters that the API will support are described in the table below. Query string parameters will be used so that users can input a period of interest, key terms, locations and time zones.

Query string parameter	Required/Optional	Description	Type
start_date	Required	The start date for the period of interest for disease reports.	String with format "yyyy-MM-ddTHH:mm:ss".
end_date	Required	The end date for the period of interest for disease reports.	String with format "yyyy-MM-ddTHH:mm:ss".
key_terms	Optional	If included, one or more key terms separated by commas. This retrieves disease reports with key terms found.	String (case insensitive and key terms separated by a comma if more than one are supplied e.g. ebola,coronavirus)
location	Optional	If included, one location only. It can be a city, state or a country. This retrieves disease reports within the locations.	String (case insensitive)
timezone	Optional	Users can choose GMT or CET timezone. Default is no specific timezone	String (case sensitive)

2.2 API MODULE RESULTS

The results found by the module will be collected once a suitable HTTP call is made. Since the disease reports will be stored within tables in the database, the Python scraper will update and edit the database daily. This will improve the speed efficiency of the API since all or most of the data will already be stored and indexed. Once a HTTP call is made to the module, the Python scraper will check for and add any news reports that may have been added in between its daily scraping. According to the parameters the user has inputted, the data will be fetched from the database using suitable SQL queries. If location parameters are given, the GeoDB API will be used to find any cities and/or

countries within the parameters given, and all of the related locations will be used in the SQL queries to find all news reports within these areas. The results will then be converted to the JSON format in the structure detailed in the project specification. Finally, this API response will be documented in Swagger showing the response headers and body with appropriate status code responses. Due to the nature of REST API's, industry standard status code responses will be shown to inform users of the result of the response.

2.3 EXAMPLE HTTP CALLS

An interaction with the API module using a HTTP call generally has the structure of the API URL with endpoints followed by the query parameters. Using the Heroku platform, this provides the ability to create a URL which will be as follows, 'http://teletubbies-who-api.herokuapp.com/'. The endpoint that will be used to retrieve disease reports will simply be called 'report'. Then, the parameters will follow, and multiple examples of different inputs have been displayed below with the start and end dates being 2020-03-06T08:08:08 and 2020-03-08T08:08:08.

Description	HTTP call
No optional parameters: all articles between the start and end date are found with no filtering.	http://teletubbies-who-api.herokuapp.com/article?start_date=2020-03-06T08:08:08&end_date=2020-03-08T08:08:08
A key term parameter: articles between the start and end date filtered by the keyword.	http://teletubbies-who-api.herokuapp.com/article?start_date=2020-03-06T08:08:08&end_date=2020-03-08T08:08:08&key_terms=ebola
A location parameter: articles between the start and end date filtered by the location.	http://teletubbies-who-api.herokuapp.com/article?start_date=2020-03-06T08:08:08&end_date=2020-03-08T08:08:08&location=France
A time zone parameter: articles between the start and end date with that time zone.	http://teletubbies-who-api.herokuapp.com/article?start_date=2020-03-06T08:08:08&end_date=2020-03-08T08:08:08&timezone=GMT
Multiple key terms: Articles between the start and end date filtered for any article that contains at least one of the key terms	http://teletubbies-who-api.herokuapp.com/article?start_date=2020-03-06T08:08:08&end_date=2020-03-08T08:08:08&key_terms=ebola,virus
Location and multiple key terms with the start and end date filtered for any article that contains at least one of the key terms in that location	http://teletubbies-who-api.herokuapp.com/article?start_date=2020-03-06T08:08:08&end_date=2020-03-08T08:08:08&key_terms=ebola,virus&location=China

Time zone, location and multiple key terms with the start and end date filtered for any article that contains at least one of the key terms in that location of that time zone	http://teletubbies-who-api.herokuapp.com/article?start_date=2020-03-06T08:08:08&end_date=2020-03-08T08:08:08&key_terms=ebola,virus&location=China&timezone=CET

Updates to API Development since the last deliverable include adding example calls and outlining parameters for time zone queries, changing the format for dates, and updating example calls when given empty parameters.

3.0 WEB APPLICATION IMPLEMENTATION

For the development of our application we are going to be using various implementation languages, development and deployment environment and specific libraries as shown in the web stack below:

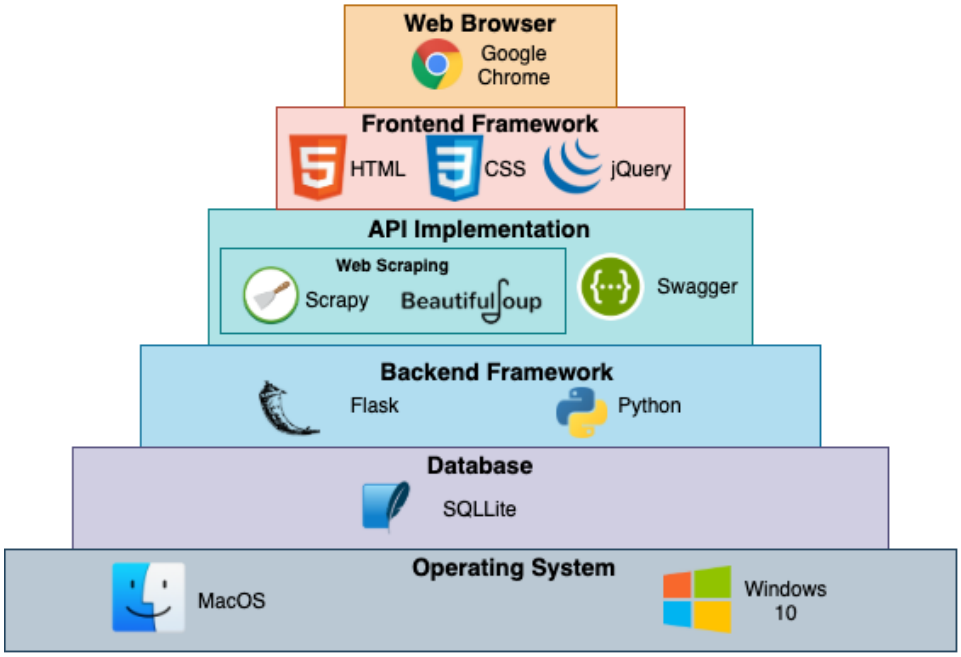

Fig 4.0 – The web stack

Beautiful Soup was added to the general web stack, with additional Python libraries and modules used explained later in 3.3 Backend Framework and API Development.

3.1 DEVELOPMENT AND DEPLOYMENT ENVIRONMENT

For the development of our web application we will be using the MacOSx Operating System and Windows 10 as they are the most frequently used operating systems. The programming languages such as Python and JavaScript which are going to be used for the implementation of the web application are fully compatible with both operating systems thus making it a good environment for development. Google Chrome web browser will also be used to access our web application. We will

also be testing our application and our API across multiple platforms such as Windows, Linux etc throughout its development to test whether our application is working successfully.

All information remains the same as previous deliverable.

3.2 DATABASE

SQLite is going to be used for the database of the web application and is going to be used to store the information received from the API. SQLite is a very light weighted database, so it is easy to use it as an embedded software with multiple devices across different platforms. It has a very good performance and can be used to read and write data very quickly as it only loads the data which is needed, rather than reading the entire file and holding it in memory. SQLite queries are smaller than equivalent procedural codes so, chances of bugs are minimal. Thus, SQLite is going to be by the team for the web application database.

All information remains the same as previous deliverable.

3.3 BACKEND FRAMEWORK AND API DEVELOPMENT

For the API development and the implementation of the backend framework we are going to be using Python and Flask. We are also going to be using SQLite for the database and using Swagger and python libraries such as Scrapy for the development and maintaining of the API.

We are going to be using Python as it is a high-level object-oriented programming language with multiple resources and libraries available online. Python has good compatibility with Flask and is used to successfully build APIs. Python is very good at error handling and has good performance across multiple devices platforms. Due to the team members familiarity with the language through past experiences we believe that Python would be well suited for the development of the backend framework and the API.

Flask is a micro web framework written in Python which works as a webserver and allows in an easier redirection of web pages. Flask is easy to write, maintain and scale web applications and can allow users to successfully build APIs. Flask helps us to route requests to the appropriate handler and allows URLs to change without having to change the code. Due to flask being fully compatible and written it Python it can be used across multiple platforms and thus has been chosen by the team to be used for the development of the API and the backend framework.

Scrapy is a free and open-source web-crawling framework written in Python. It can be used to extract data using APIs or as a general-purpose web crawler. It is compatible across multiple devices and can run on various platforms. It processes and schedules the HTTP requests asynchronously and thus it can take care of multiple requests at a given time. This allows for other requests to continue even if there is an error that occurs while some request is being handled enabling us to make it suitable for API development for our project.

Additional libraries and modules used to help web scraping since last deliverable include: Beautiful Soup, Pytz, Text2Digits, Urlopen, Pycountry, Geotext and Unicodedata.

Python Library / Module	Use	Benefit
Beautiful Soup	In linkbot and updatebot (used to traverse links and update database regularly). In reportbot, used to grab first paragraph of text.	Provided more useful functions for separating maintext, removing encoded and special characters and hence we were able to write more concise / readable functions
Pytz	To check if regex expression matches a valid time zone	Provides a database of accurate time zones, allows time zone conversions / calculations
Text2Digits	Given a text, converts all numbers written in word form into digit form (e.g. three -> 3). This was used to find number of cases and deaths in reports.	Information on reports could be processed more quickly so less functions were needed to be written manually to convert text to the same format to find cases and deaths
Urlopen	Fetches url to be used by Beautiful Soup	Works well in conjunction with Beautiful Soup with lots of documentation and tutorials
Pycountry	Used to check if a country/state is mentioned in text to find locations in reports	More comprehensive database of locations and already consists of functions which link states to countries.
Geotext	Used to check if a city is mentioned in text to find locations in reports	Pycountry lacked database on cities, and Geotext had functions linking cities to countries.
Unicodedata	Used to remove letter accents in location names to compare and search accurately	Complete database of Unicode characters with functions involving NFKD which can normalise characters

Overall these libraries and modules were chosen because they were quick to import/install and learn how to use, as well as provided a wide database of information and functions.

3.4 FRONTEND FRAMEWORK

For the implementation of the frontend framework we will be using JavaScript (jQuery), HTML and CSS.

We will be using HTML and CSS as they allow the information on the websites to be displayed in a systematic manner and allows us to develop our own layout for the website. CSS allows us to make our web application more aesthetically pleasing. HTML works cohesively with Flask which is going to be used in our backend implementation and thus making it a good resource for implementation.

We will be using JavaScript for frontend development as it allows the webpages to be more aesthetically pleasing and dynamic. JavaScript also has good compatibility across devices and works

well on different operating systems and browsers. It can also quickly interpret and run the code which leads to less runtime errors. It supports multiple modules and there are multiple resources available to learn JavaScript from thus making it a good language for frontend development. As jQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish and wraps them into methods that you can call with a single line of code, we will be using that library for our code implementation.

All information remains the same as previous deliverable.

4.0 CHALLENGES AND SHORTCOMINGS DURING DEVELOPMENT

4.1 WEB SCRAPING

Splitting articles by disease reports was a challenge as articles sometimes mentioned more than one disease or event and information would be scattered throughout the article rather than all in one paragraph. Hence, we had to find diseases mentioned first and split the article by each disease. Some diseases had multiple names/aliases such as Coronavirus/COVID-19 or Influenza types which required extra scraping and filtering.

Separating main text and removing encoded and special characters e.g. '\n' was difficult using Scrapy, which is why we used Beautiful Soup functions for this section. For syndromes, prior research was required to find common symptom words and match them to diseases and then linking them to the correct medical term outlined in specifications e.g. diarrhea and vomiting were matched to acute gastroenteritis. Event Date was difficult as it was a string consisting of both words and numbers and had to be converted and sorted for searching by range of dates. Challenges with location included finding an accurate database with correct linking between cities and states and the countries they resided in, which was overcome by using Pycountry and Geotext. Cases was also difficult since sometimes information was stored in tables but there was no set format/class name for all tables.

Overall, there were a tiny number of URLs which took too long to process or timed out, and due to the large amount of filtering and processing for each URL, scraping the entire database in the beginning once took extremely long.

4.2 DATABASE MANAGEMENT

Scraping daily required finding a way to run updatebot daily, requiring a lot of in-depth research. Cron was used first but errors occurred as the files weren't within the home directory and could not be accessed easily when within completely different directories. After trial and error, we found the best way without experiencing drifting or using a sleep timer was by using Apscheduler and Cron together which allows updatebot to check for new reports. However, this had threading issues with Scrapy and was incompatible with Flask, resulting in the database not being updated.

4.3 API DEVELOPMENT

Our database design involved many tables and considerably long queries which required many joins to be written in order to connect information and obtain the correct data. Otherwise, there were no issues.

4.4 WEB SERVICE MODE

Due to the structure of GitHub repositories with nested folders, the API could not be run properly within Heroku. To resolve this, files related to the development of the API were moved to another repository with a simple file structure where the main app located in the root folder as seen in https://github.com/sarahoakman/who-api-web-service

As mentioned before in 4.1 Web Scraping, daily scraping could be accomplished through Cron or Windows Schedule, but this didn't update the database in the server on Heroku. Heroku Scheduler addon was used to create a schedule to run updatebot at 12am each day. During testing, we found that updatebot runs and shows correct output when scheduled for every 10 minutes but doesn't update the database. This is due to Scrapy Pipeline not working when deployed on Heroku. As a result, we tried to manually add to the database by grabbing the output from reportbot and passing it directly to the database adding function in pipeline.py.