Deliverable 2: Updated API Design Details Report
SENG3011: THUMBNAILS

Zixiang Lin z5314168
Eric Phung z5234001
Bavan Manamohan z5208542
Kaaviya Salai Manimudian z5260340
Nitharshni Chennai Kumaravel z5255563

Table of Contents

API LINK:

https://flask-service.boulmkfsb234o.us-east-1.cs.amazonlightsail.com/

Final Architecture of API and Justification of Choice of Implementation

Firstly, we have implemented a Pythonic scraper that will access the following website:https://www.cdc.gov/outbreaks/ and navigate to specific outbreak articles. Within these articles, the scraper parses the HTML code and collates relevant disease and publication information to store in a NoSQl database. Everyday at a set time, the scraper runs again to ensure that the data extracted from the website is timely, relevant and concurrent with the data already existing in the database. This ensures that if there are updates detected on the CDC website then the relevant fields in the database are updated before the API accesses them.

We followed the REST architectural style to build our API, which allows us to make use of the 'HTTP GET' command to retrieve data from the CDC website with inputs entered by the user. When requests are made by the user through the API with either the Swagger UI or URL interactions , a representation of the state of the resources collected in the database is transferred to endpoints via JSON. For this assignment we decided that having the API only communicate with the MongoDB database through the Pymongo library would allow for the overall structure of the system to be coherent as well as simplify the development process.

HTTP Method	API endpoint	Parameters	Description
GET	/findstart_date=<date1>&end_date=<date2>&location=<location>&keyterms=<term1,term2>	Start date End date Location - *(optional)* Key terms - *(optional)*	Returns all reports within the date range and location or/and key terms (if they are provided)
GET	/findAll	*No parameters*	Returns all reports within the database
GET	/find/location<location>	Location	Returns all reports containing the

			searched location
GET	/find/keyterms<term1,term2>	Key terms (separated by a ‘;’)	Returns all reports containing the searched keywords
GET	/find/datestart_date=<date1>&end_date=<date2>	Start date End date	Returns all reports within the date

Table 1: Updates API endpoints

Justification of Change of Database

In this project, we had decided to use a database that would store all the values that the scrapers would collect in order to develop our API. Initially we had planned to use a database in SQLite, however after some consideration we have decided to use NoSQL, particularly MongoDB. The main reason for this is that the relational structure of SQLite tends to restrict how tables should be designed. The data our program will be receiving will contain multiple attributes such as location, diseases and syndrome which are in arrays, which is difficult to store with required schema into a SQLite database. Additionally, maintaining multiple tables, each connected with primary keys and foreign keys and using this to search through with queries would have made the search process very long and tedious.

MongoDB on the other hand uses document schema, which provides much more flexibility to store and model data structures. Due to this, we can now easily store the json objects created by the scrapers as documents into the database, thus allowing for efficient storing and search processes. Also, it is supported by AWS, which is the (VPS) provider we have decided to use for this project.

Other Important Libraries

Library	Justification

Flask 2.0.3	Allows for easy development of a web application using python and request dispatching of our database using REST.
Flask-Restx 0.5.1	Restx is the best actively maintained extension for Flask that added support for building REST APIs in Python.
Pymongo 4.0.2	Facilitated integration with MongoDb. Was used to connect to the database and collection as well as query results.

Table 2: Important Libraries used

Challenges and Shortcomings

There were many challenges faced during the implementation of the API. One of the major obstacles was the structure of the website we were given to scrape. Each page on the CWC website that contains the details of the different diseases follows a different structure in HTML. For example, the Ebola pages will have a completely different structure to the pages with information on Listeria. Due to this, it was not possible to create just one scraper that will work for all the pages we were required to scrape. After some research and discussions with our tutor, we were advised to make a scraper for each type of disease, which ended up being extremely time consuming and stressful.

Also within each of the sub-pages of each individual disease itself, the structure and formatting of the pages varied. There were also many pages that contained collapsable tables with required information stored in there as data alone or as links that needed to be opened. This however was very difficult to extract using BeautifulSoup (which was what we had decided to use to create the scrapers) as this information would not be parsed through it. Long periods of time were spent looking to see how to extract information from the collapsible tables through BeautifulSoup, but the methods found proved to be extremely difficult and eventually unsuccessful at providing it. Thus, this section had to be left out for this deliverable.

Moreover, when we tried to extract the main text from each page, we noticed that there were multiple container boxes containing text on certain pages and

only one on others. This proved really difficult to collate all the text together. Thus when possible, the paragraphs were appended together or the paragraph with the most information was considered the main text. Other times there would be hyperlinks within the paragraph that distributed the appending process, which thus inhibited other data retrieval processes linked to it (this can be seen in the Ebola scraper). Hence in certain places, due to time limitations they had to be hardcoded. We are aiming to rectify this by the next deliverable.

When utilising the diseases and syndromes json files provided for us, we noticed that the diseases.json had references to “other” and “unknown”. When this was used in the scraper, it picked up on words that weren’t diseases, so they had to be removed from the json.

Furthermore, our team implemented a script that executes scraper automatically every 24 hours. We used AWS lambda function to run the scraper, and AWS EventBridge that creates a clock to execute the lambda function once per day. This allows our database to be automatically updated everyday and ensures all the sources are updated. The lambda function is located at lambda_function.py.

```python
import re
from pymongo import MongoClient
from listeria import listeria_scraper
from ebola import ebola_scraper

def lambda_handler(event, context):
    cluster = "mongodb+srv://thumbnails:thumbnails@cluster0.lfkm3.mongodb.net/SENG3011?retryWrites=true&w=majority"
    client = MongoClient(cluster)

    # Select the database and the cluster
    db = client.SENG3011
    collection = db.SENG3011_collection

    # def clear_db():
    result = collection.delete_many({})

    # def update_dp():
    results = []
    for x in listeria_scraper():
        result = collection.insert_one(x)
    for x in ebola_scraper():
        result = collection.insert_one(x)
```

Figure 1: Displaying Lambda Function

Updated Gantt Chart

T17 thumbnails'
Project GANTT CHART

Report Design Details 1*
- Describe API development
- Explain Web service mode

Report Design Details 2*
- Explain API parameters
- Sample URL

Report Design Detials 3*
- Justify implementation language,development and deployment environment, libraries