Design Details

Description, Justification, Challenges and Shortcomings

GraphQL

At the suggestion of our mentor, we create a GraphQL API rather than a REST API. Instead of using traditional POST, GET, PUT and DELETE endpoints to create, read, update and delete, users instead send queries, mutations and subscriptions in a JSON format to a single POST endpoint to do the same.

As GraphQL is more expressive than REST, clients can perform the same tasks using fewer calls to the server. However, as the specification was written for a REST implementation, we needed to make several compromises.

GraphQL does not have a direct equivalent to Swagger for documentation and interactive testing. Most GraphQL packages are bundled with an interactive GraphQL client such as *GraphiQL*. However, these clients are typically not enabled in production and document the API in a terse manner. It was thus necessary for us to direct users to use an interactive GraphQL client-hosted elsewhere for interactive testing and to provide more readable examples in our Readme.md.

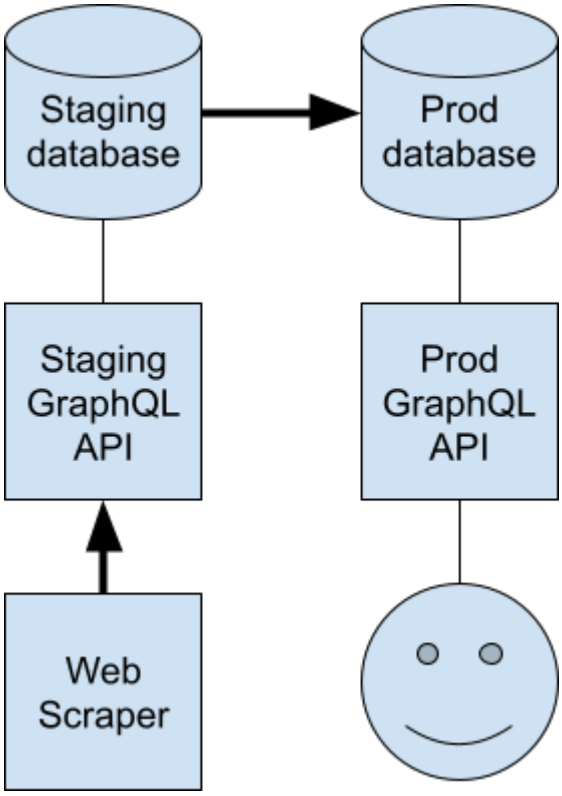
AWS Amplify

Also at the suggestion of our mentor, we used AWS AppSync to deploy the GraphQL API. More specifically, we used AWS Amplify to create a complete backend including a database on AWS DynamoDB, authentication and the GraphQL API using AWS AppSync.

By using AWS Amplify, we were able to interactively create the entire backend. Using the Amplify Admin UI, we were able to specify the database schema using a GUI which automatically creates a corresponding GraphQL API with functions for querying, mutating and subscribing to changes in the database. We are also able to create, read, update and delete rows in the database using the GUI, simplifying management.

Furthermore, using the AWS Amplify CLI, we are able to test a mock of the entire backend on our local machines with a mock SQLite database and a locally compiled and hosted GraphQL API. A *GraphiQL* client is also made available which allows us to interactively create GraphQL queries.

Using the AWS AppSync console, we were also able to easily change the API key and expiry for the GraphQL API. We were also able to easily enable logging of API calls to AWS CloudWatch, where logs can be easily filtered and statistics displayed graphically. This made fulfilling the logging requirement of the specification simple.



System Diagram

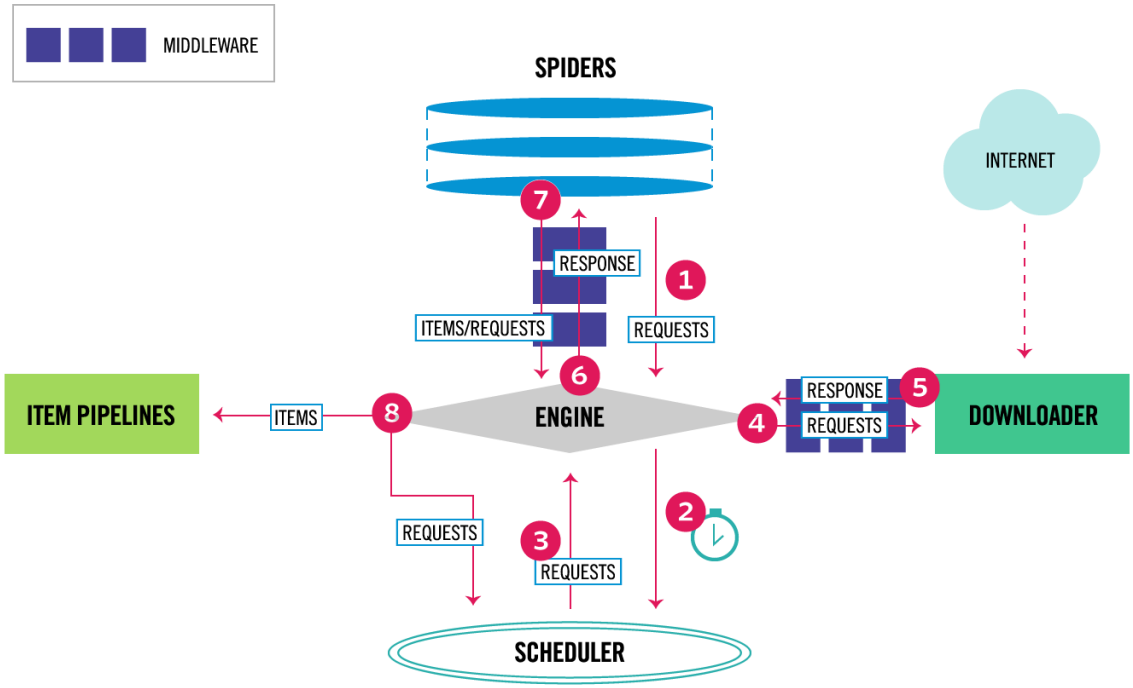
However, AWS Amplify and AWS AppSync are highly opinionated APIs and not without bugs. Firstly, it is not possible to create a GraphQL API without any form of authentication. This means even users of the production API will need to use an API key to access it. Furthermore, limiting users of an API key to only be able to query, but not mutate or delete entries through the Amplify Admin UI or AWS Amplify CLI does not work. Using the other authentication options, Amazon Cognito or AWS IAM, would have required a complex authentication flow only realistic for users if called from a frontend using the corresponding packages for web or mobile clients. Since we expect users to call the backend using simple requests such as through Postman rather than using a frontend, these were not options.

Instead, we resorted to creating a production backend that has the mutation and subscription functions deleted through the AWS AppSync console. This means data from the web scraper must first be uploaded to the staging deployment using GraphQL mutations, then copied using a NodeJS package such as “copy-dynamodb-table” to the production deployment. This is acceptable as data from the web scraper will only be uploaded once. However, were the web scraper to be run automatically, this is clearly undesirable.

Finding a solution for the deficiencies in AWS Amplify likely took more time than writing our own GraphQL server using a framework such as Apollo which also allows enabling an interactive *GraphiQL* client in production. AWS Amplify and AWS AppSync are thus clearly targeted at web app developers rather than API developers and should not be used for the latter.

Web Scraper

Scrapy



Only one member in our group has previous web crawling experience. *Scrapy* is an efficient and easy-to-use tool, so we used it as a way to extract data from websites. We can easily extract data using the Scrapy selector by processing the HTML tags. For instance, we can get the headline of each article using the *<title>* tag. However, even though *Scrapy* is an easy-to-use tool for beginners, we still encountered a challenge when trying to clean the text of the main content. The articles on the "*Outbreak News Today*" website have different structures. Some articles have the main content under the *<p>* tag, while others use different tags. Some articles have dot points in the main content and other irrelevant texts also use dot points or the *<p>* tag. This took us a lot of effort to solve. Since the date of publication of an article is usually close to the event date, some articles may not contain the date when the event happened, in these cases, the event dates are set to the date of publication.

SpaCy

For the data in the reports list, the diseases, syndromes, and locations are extracted and classified from the headlines and the main texts. Originally, we used *SpaCy* as an NLP tool to classify them. However, diseases and syndromes are both labelled as "DISEASES" in the SpaCy corpus, which makes the classification process more difficult. For the syndromes part, we still used a combination of *SpaCy* and some hard coding for the part-of-speech and the lemmatization tag part to obtain the desired result.

For some unknown reasons, *MyMerpy* did not work on our local machines, but it worked on the CSE Vlab. Memory use increased when we ran the program to process texts using SpaCy, and therefore the program was killed because of the limited memory usage in the CSE environment. We also considered using the *AWS Comprehend* to classify the

syndromes, but the database is incomplete. For example, COVID-19 was not in the database package, and we could not add it manually either. So we went back to using *SpaCy* and found that we only need to collect the garbage after each text processing, so the memory usage issue was fixed.

MyMerpy

Merpy is a python package that wraps around *MER* - the Named-Entity Recognition tool developed and maintained by lasigeBioTM. Given any lexicon and any input text, *Merpy* is able to return the list of terms recognised in the text, including their exact location (annotations) in the original input. We have forked the *Merpy* repository and renamed it as *MyMerpy* to extend the functionalities and make it more suitable for this project.

The medical corpus available for *SpaCy* is not complete or up-to-date, this has caused an issue with recognizing newly appeared diseases and differentiating disease names from syndromes. *MyMerpy* takes the advantage of the *Human Disease Ontology*, which is a comprehensive hierarchical controlled vocabulary for human disease representation and tweaks it to output a few compact lexicon files. With *MyMerpy*, we are also able to add diseases manually into the lexicons as it gives additional scalability for modification.

Deep down inside *MER*, it was developed and tested using the GNU stream editors including *awk (gawk), sed and grep*. There has been an issue with BSD version command line tools on *macOS* such that the program is not assured to work on macOS without installing GNU utilities.

Another potential difficulty of using *MyMerpy* is that disease names mentioned in the articles can be abbreviated, such that they are different from the medical terms in our lexicons. There has been a solution provided by lasigeBioTM that returns all possible terms that are similar to the recognized ones. We will look into this solution in the next deliverable as it's not a vital problem for the current deliverable.

Geonamescache

For the location part, *SpaCy* identifies both countries, cities and states as "GPE" and has no distinction between them, therefore we had to resort to other libraries to further categorise places names in GPE into country and location. We also considered *AWS Comprehend*, but it also does not differentiate between country and other types of locations. Therefore, we initially used the *Wikipedia* API, the definition of the obtained location name on Wikipedia will be checked, if the keyword "country" exists it will be classified as a country and location otherwise. But this approach did not achieve desirable accuracy either, for example, Norway could not be identified but "or way" was identified instead. We also tried using *Nominatim* API, but as there is a limit to the request number we can make, we did not use it in the end. In the end, we used *geonamescache* to extract the country and the city as it can extract the country and the city separately. Every city is given a related country code, which can be used to link the country to the city.
However, the database of *geonamescache* is also incomplete. For instance, Västra Götaland is a county in Sweden, but it is not inside the database. Then, we decided to use another Python library called *geograpy*, since it does contain the counties in their database. However, *geograpy* can only output countries, cities and regions. Regions contain all the

data of the country, the city, the county and other stuff (such as diseases). It doesn't have a method to get the county, and some diseases name are being classified as regions, e.g. "Yersinia". Hence, we continued using *geonamescache*, even though the database is incomplete, but it works well for now and this package also contains the geonames_id which will help us to do the advanced version of the locations part. However, we will add this in the next deliverable.