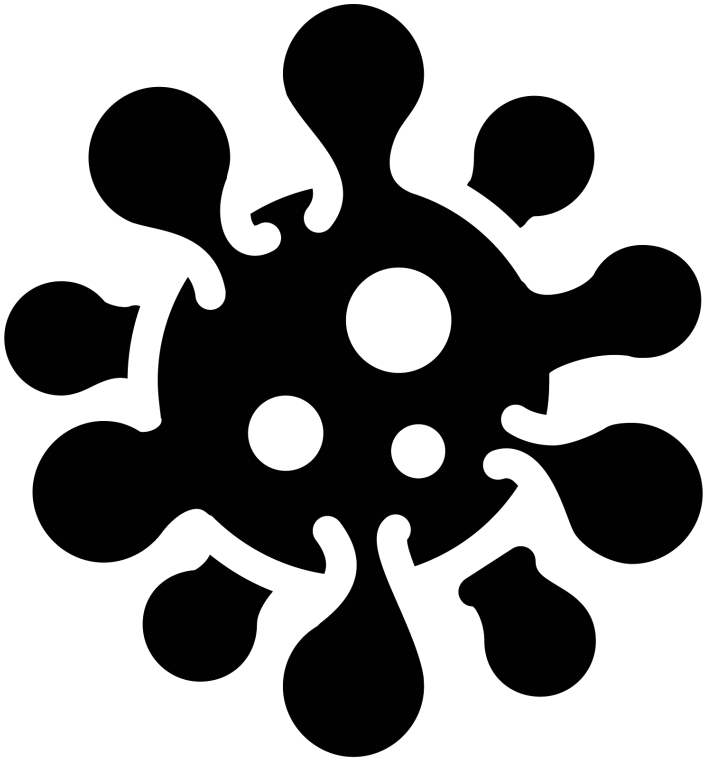
Design Details Report



Team Viral

Aimen Hamed - z5310143
Henry Ho - z5312521
Leila Barry - z5206290
Neelam Samachar - z5310063
Shannen Jakosalem - z5164261

Mentor: Nikhil Ahuja
Course: SENG3011

Table of Contents

Introduction

The purpose of this report is to elaborate how Team Viral's API is being built. The team has varying levels of understanding as to how the API works due to the diversity of our technical experiences and, given that the API is primarily backend, the frontend team may have limited exposure to it. This document therefore serves to foster a shared understanding of the API being built as well as communicate design details to our mentor for the purposes of deliverable 2. As the team branches off into different technical specialisations such as the crawler and frontend, this document will help us put our work into context and understand how each team member's assigned parts will interact with those of others.

Since the first deliverable, there have been no changes to our software architecture although we have faced issues with our database deployment and are currently researching for potential alternatives to Heroku. The details of this are discussed below in our Challenges in Implementation. If a change is required this will be discussed in detail in the next deliverable's report alongside the process we followed to ensure the change causes minimal disruption to the work we have completed thus far.

This report is divided into three sections that touch on a particular aspect of our design and implementation. These sections, what they cover and why they are discussed are presented in the table below in order of when they will appear in the report.

Section Name	**Description**	**Purpose**
Tech Stack	An overview of the technology the team will be using and a justification of why we have chosen such technology.	To provide a high level view of what our API looks like under the hood.
Advanced Logging Capabilities	A discussion of how we have implemented logging capabilities in our API.	To provide context and justification for our choice of advanced logging capabilities.
Challenges in Implementation	An overview of the challenges we have faced in our implementation to date.	To communicate the challenges we have faced and our methods of resolving these issues.

Tech Stack

Software Architecture Diagram

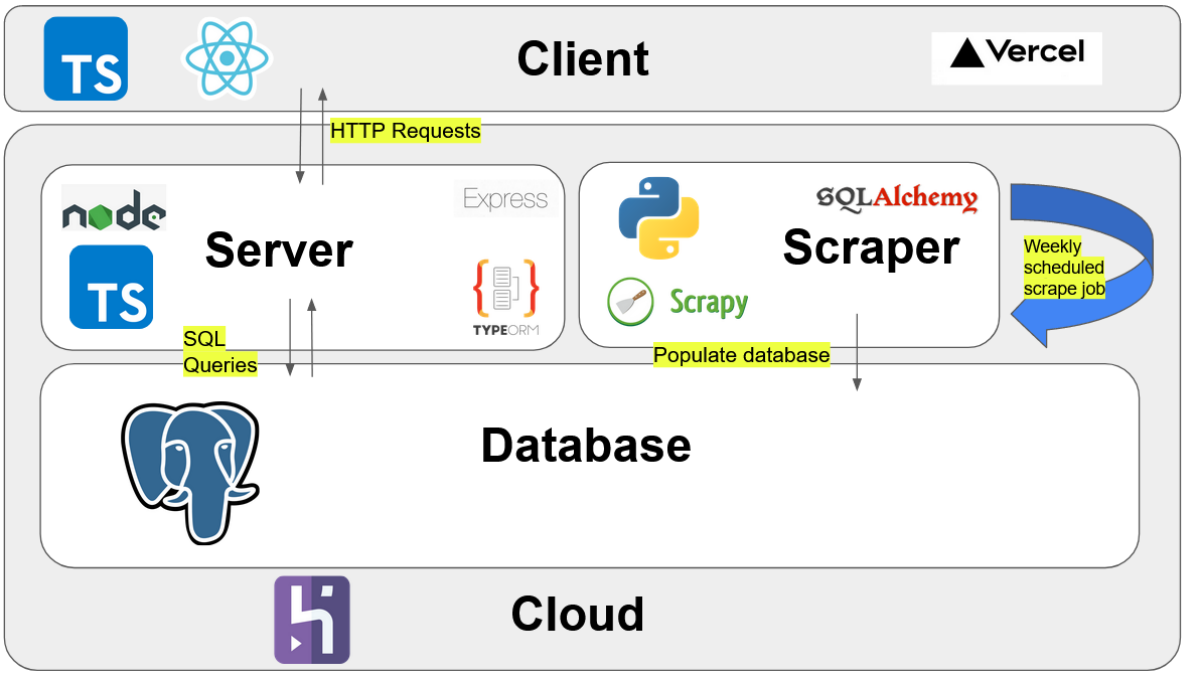


Figure 1. Visual diagram of the tech stack

Frontend

Requirements

Our frontend is the only interaction that users experience directly, and as a result,it is one of the most important aspects of our application. The application's frontend must be:

- Aesthetically pleasing
- Simplistic, minimalistic and modern
- Effective and useful
- Intuitive and easy to use

The frontend must also possess the capability to make HTTP requests to our backend server, and handle large objects and data which can be processed into UI elements. A powerful frontend is crucial to the functionality and aesthetic for our application.

Researched Choices

To handle this type of logic whilst also providing an aesthetically pleasing UI, **JavaScript** is the most intuitive language to be writing our frontend in. More specifically, our team will be using **TypeScript** to enforce type declarations to ensure more readable code within our codebase, whilst also providing better alignment with the expected interfaces with our backend API responses and requests.

From our decision, we still needed to decide which framework we were going to use in conjunction with TypeScript for our frontend application.

React is an obvious and popular option to consider to be building our frontend application, being supported by a huge community with lots of useful guides and documentation. As a result of the popularity of the React framework, there are many useful tools which are designed to be utilised with React such as NextJS and Redux which offer great utility for our frontend capability. Along with these advantages, React is also where our team has the most experience and knowledge, becoming a much easier pick for our stack.

Benefits	**Drawbacks**
+ Well supported and documented + Most popular JS frontend framework + Works well with other utility libraries + Powerful functionality and toolset	- Can be considered bloated - Lots of boiler code

Although React was almost an instant selection for our frontend framework, we also considered another noteworthy framework, **Vue**. Vue packs a lot of the same advantages that come with React, having the same capabilities to implement complex and useful frontend features. However, the main call to not use Vue was the steep learning curve, as its syntax is quite drastically different to the likes of React.

Benefits	**Drawbacks**
+ Well supported and community driven + Popular JS frontend framework + Powerful functionality and toolset	- Steep learning curve - Not as well integrated with other utility tools - No experience in team with framework

A third choice we considered to develop our web application was **Angular**. Compared to React, Angular is a full-fledged framework with all of its necessities built into it that is built to be written in TypeScript. However, similar to Vue, we found it difficult to justify using Angular due to its innate complexity. Coupled with our team's lack of experience working with this framework and the rapid pace of development we would need for this project, we concluded that this choice is considered unviable.

Benefits	**Drawbacks**
+ Built on Typescript + Easy to start out of the box + Full framework.	- Steep learning experience - No experience in the team with the framework.

Conclusion

For this project, our team has decided to use React, as our team is able to iterate and develop our frontend application efficiently with our prior background knowledge of the framework. In conjunction with React we plan to use TypeScript to improve our codebase clarity by enforcing type declarations on component props, as well as backend requests and responses which will ease the complexity in development. Along with this, React has an important advantage of having strong compatibility with Redux, a global state container, which isn't as strongly available or supported in other frameworks such as Vue. Finally, React is the easiest framework to pick up with the vastly large support and tutorials online for a lot of use cases, which will make the development experience for all members of our team more efficient.

Frontend Deployment

Our frontend application will be deployed via Vercel, as it offers a very intuitive and easy to use tool to deploy the frontend, especially the React apps. Vercel has great integration with Github, and allows deployment and builds straight from a branch on the repository. It also provides access to the build logs and configurable build times, which is useful for our mono-repo structure as instructed to us by the specification.

Backend

Requirements

The specification provided instructs at least one API route which accesses a search service which can be filtered by a criteria. In order to handle these requests our backend service must:

- Have defined routes with a HTTP method *(POST, GET, PUT, DELETE)*
- Middleware validation on request fields
- Access and query database
- Filter and handle business logic
- Be scalable under heavy load / handle traffic

Researched Choices

Express is a minimal and flexible NodeJS web application framework that provides a robust set of features for web applications. It is one of the most commonly used frameworks for server operations with JavaScript, and has extensive functionality through community support, with additional utility tools such as NestJS. Express is tightly coupled with NodeJS, meaning it benefits from its performance, but also creates extra overhead in maintenance of the codebase with package management.

Using Express also means that the backend would be written with JavaScript, which is notorious for unexpected behaviour, introducing strange edge cases within our API service. However, JavaScript does have advantages, especially with asynchronous programming allowing concurrency with promises.

Benefits	Drawbacks
+ Minimal and flexible + Robust set of features + NodeJS performance + JavaScript asynchronicity + Same language as frontend	- NodeJS maintenance overhead for packages - JavaScript inconsistencies and strange behaviour

Flask is another minimal web framework that uses the Python language to provide a lightweight backend service for web applications. Similarly to Express, it is much simpler than the alternatives such as Django, allowing for flexibility in its usage. There are third party components which can be used with Flask to build a backend service, but they are quite limited and don't offer complete utility for features such as schema validation. Flask being used with Python is easy to write, but very difficult to maintain and create predictable code, since it is a loosely typed language, leading to unexpected behaviours. Whilst Python doesn't have promises, it does offer multi-threading, allowing code to be executed in parallel.

Benefits	Drawbacks
+ Minimal and flexible + Essential features only + Easy to write + Multi-threading	- Python, loosely type which can cause issues in group work - JavaScript inconsistencies and strange behaviour - Different language to frontend

Java Spring Boot is another popular alternative for writing backend software. With strongly typed variables and muilti-threading built in, Spring boot is a popular, long-standing framework and typically sees use in large enterprise stacks where maintainability and performance is key. What makes this an unfavourable choice is once again, the pacing we require to complete this project. Rapid development requires strong knowledge and experience, which none of our team members have. Compounded by the lack of familiarity with Java, we considered this choice unfeasible.

Benefits	Drawbacks
+ Well established framework. + Multithreaded Capability. + Large community behind it.	- Multiple people in the team are unfamiliar with Java. - Also a different language to front-end.

Conclusion

Our team decided that we are going to use Express and NodeJS as our backend framework. Express is lightweight and flexible to fit for our use case in making an API service, and comes with the benefit of having the ability to integrate with other utility libraries. Specifically, TypeORM is what we plan to use with our Express server to save data and query the database, allowing explicit type and relational declarations so that our backend can correctly reflect the database structure. Furthermore, the major contributing factor to our selection is that we can use TypeScript with Express so that we can benefit from the framework's advantages whilst staying type safe. Another major benefit in this decision is that the language is the same as the frontend, meaning that we can share interfaces for what's expected between API requests and responses allowing easier development and alignment. Along with this, the transition and handoff for team members to move to the frontend and backend is less significant, improving our work efficiency.

Backend Deployment

Our backend application will be deployed via Heroku, as they offer a nice suite of app deployment options which can prove useful for the remainder of our stack. Heroku has a dedicated CLI tool to deploy applications via Git versioning, which is well documented and appropriate for our use case. Once deployed, Heroku provides us the API URL which our frontend and others users can access our backend endpoints with. Heroku also has lots of configuration options such as which port we expose for our service, which is useful to have defined.

Scraper

Requirements

As per the specification given, our API's information source must be scraped from the website: https://www.who.int/emergencies/disease-outbreak-news. To collect all available information from this web page, our scraper must be able to:

- Scrape raw data from each article listing.
- Process this data into usable information.
- Navigate through the webpage through pagination and following links to the article.
- Communicate with a server or database for persistent storage after a successful scrape.
- Automated crawling on a scheduled basis. (e.g. once a week)

Researched Choices

A worthy mention that introduces many people to web scraping in Python is **BeautifulSoup,** as it introduces all the basic tools that you would need to scrape through a given webpage. However, BeautifulSoup is less a crawling framework and more a utility tool as it lacks many tools necessary to crawl through a website. One of its key limitations is its inability to scrape more than one page which eliminates its potential for this project.

Benefits	**Drawbacks**
+ Introduces webscraping's basic tools.	- Cannot crawl through multiple websites without external support.

Scrapy is a complete Python framework that excels in extracting information by crawling through websites and immediately processing it through its built in pipeline support. It is designed for rapid crawling through its multithreaded requests and can extend its functionality through extensive community support's ability to add in middleware and plugins. However, it cannot easily scrape Javascript heavy pages such as AJAX pages, as Scrapy does not call any Javascript (although this can be circumvented through other libraries).

Benefits	**Drawbacks**
+ Rapid asynchronous requests. + Lots of community support through tutorials, middleware and plugin extensions. + Tools exist that automate crawling through proxies.	- Cannot handle JavaScript heavy websites (e.g. sending AJAX Requests). - Non-beginner friendly.

Selenium is another Python framework used for web testing and automation that can be adapted for web scraping as it also possesses the means to view and interact with html elements. What makes Selenium a potential candidate for web scraping is its ability to handle AJAX pages through its headless browser which Scrapy struggles against. This

would let it handle dynamic loading where data is requested after the page has loaded. A consequence of this is that the headless browser is more resource intensive than other candidates. In addition, scraping data is much slower through its headless browser which makes it struggle when dealing with high volumes of data.

Benefits	Drawbacks
+ Can easily bypass javascript roadblocks (handling javascript elements)	- Lower performance as it uses a headless browser to navigate through pages

APIfy is another strong candidate that offers web scraping service and automation through its cloud-based platform and SDK written in Javascript. In it are a variety of different spiders that can be used, namely CheerioCrawler, Puppeteer, and Playwright which all specialise for different needs in crawling. Crawled data is then stored on the platform and can be accessed in a variety of accessible ways such as JSON, CSV, XML and more. As this platform utilises Javascript to program the crawlers, it is potentially beneficial in unifying our tech stack and removes the need to learn another language.

Benefits	Drawbacks
+ Can also handle javascript elements + Three different web crawlers that specialise in different situations. + Extensive documentation	- Cannot insert directly into a database. - Requires post processing. - May need to pay a subscription for the volume of data we are scraping

Conclusion

For this project, our team has decided on using Scrapy as our spider crawling framework. Compared to Selenium and APIfy, Scrapy is a lighter and more portable framework that appears to be more appropriate to our needs where post processing of items is required and stored externally. Furthermore, the required data within the website will not require the use of javascript to load, eliminating the need for headless browsers. To supplement this choice, a member of our team has prior experience in web scraping with BeautifulSoup, making this framework easier to work with.

Web Scraper Deployment

To connect our web scraper to the rest of our tech stack, we plan to use Scrapyd and SQLAlchemy. Scrapyd is an application that is used to deploy, control, and handle output coming from a spider on external hosting. On the other hand, SQLAlchemy will provide an external database and will allow our team to upload our scraped and processed information. This bundle will then be deployed on 2 Heroku apps: a server to hold our project's spiders, and another to host the GUI scrapydweb used to manage the spiders and schedule regular crawls.

Alternatives that were considered include a Flask application that will send an output file of scraped items to the server and an automated cron command. However, it was considered too cumbersome to maintain when deployed.

Database

Requirements

Data scraped from the website must be readily available for users of our API to use. While a class or dictionary-based approach is sufficient for small amounts of data, performance rapidly deteriorates when trying to organise and maintain information consisting of thousands of entries. As our API will be handling scraped data from a website with over 2000 articles currently, a database will be necessary for keeping this information organised and returning rapid responses to our users. Hence, the database must:

- Rapidly respond to queries.
- Strictly maintain data integrity.
- Offer access to complex data types.

Researched Choices

SQLite is a small portable DBMS that stores its data in a local file that can be placed directly inside of the app. A benefit of its lightweightedness is that it is also a good introduction to learning SQL and persistent storage, since it is very easy to set up and use due to its simplicity. Unfortunately, its simplicity is a drawback for this project as it would hinder our development when trying to write secure and maintainable systems. Furthermore, upon further research that looks into the potential to deploy this database, we found that SQLite is not well supported, if at all, making it impractical to use.

Benefits	**Drawbacks**
+ Small lightweight choice makes learning incredibly easy. + Uses well established SQL to query.	- Lack of advanced data types and features. - Cannot handle multiple concurrent users. - Cannot be deployed to servers.

PostgreSQL is a powerful open source DBMS that extends the SQL functionality while maintaining its base principles. What makes this database beneficial for us is its ability to store a wide range of complex data types compared to other SQL languages and allows users to define their own custom data types. On top of having a built-in procedural language used to create functions, PostgreSQL is also well supported through frameworks from other languages including Python (SQLAlchemy) and Javascript (TypeORM). However, as PostgreSQL is based on SQL's foundations, it would require rigid, predefined tables to store data. This would need careful planning and strong knowledge on how the data should be stored. Another issue is that while external frameworks help ease interaction with the

database, having a solid foundation in SQL syntax and functionality is required to efficiently query for the desired data.

Benefits	Drawbacks
+ Many advanced data types compared to other DBMS as well as custom data types. + Rapid query processing time. + Built-in procedural language used to write powerful functions with automated calls. + Good compatibility with many common languages.	- Large learning curve in learning the basis of SQL languages, procedural language, and more. - Necessitates planning from the start on how data is stored.

MongoDB is another alternative for our project that vastly differs from traditional databases, storing information in key-value pairs. This allows data to not be so rigidly defined, making it more intuitive and granting flexibility in storage. Querying is also easy for users who are more used to traditional coding compared to SQL queries. Unfortunately, MongoDB's flexibility in its storage does come at the cost of data integrity as it does not enforce ACID principles. This could lead to inconsistencies in the database and produce conflicting information in separate queries.

Benefits	Drawbacks
+ Simpler more intuitive syntax and storage form + Flexible storage form through key-value pairs with defined variable types. + Strong upward scalability	- Less flexible in merging different tables together. - Does not enforce ACID principles

Conclusion

Our team chose PostgreSQL to be our database with a major factor forming our decision being our experience. Most members in our team have interacted with some form of SQL database and have previously used PostgreSQL and have a strong fundamental understanding of how to use it. Furthermore, an ER diagram between fundamental objects has already been rigidly defined in the specifications given, bypassing the need to research our database's design prior to implementation. Hence, we believe that building a database with PostgreSQL would be the most suitable choice for this project.

Database Deployment

To deploy our chosen database, we plan on using Sqitch to help externally deploy our database's schemas and tables on the platform Heroku.

Database Schemas

Article

article_id	**uuid, primary key**
url	String, non-nullable
headline	String, non-nullable
main_text	String, non-nullable

Report

report_id	**uuid, primary key**
diseases	[<String>], non-nullable
syndromes	[<String>], non-nullable
event_date	Datetime, non-nullable
locations	[<String>], non-nullable
article_id	uuid, foreign key, non-nullable

User

user_id	**uuid, primary key**
name	<String>, non-nullable
email	<String>, non-nullable
password	<String: SHA-256>, non-nullable
bookmarks	[uuid], foreign key, non-nullable
dashboards	[uuid], foreign key, non-nullable

Dashboard

dashboard_id	**uuid, primary key**
user_id	uuid, foreign key, non-nullable
widgets	uuid, non-nullable

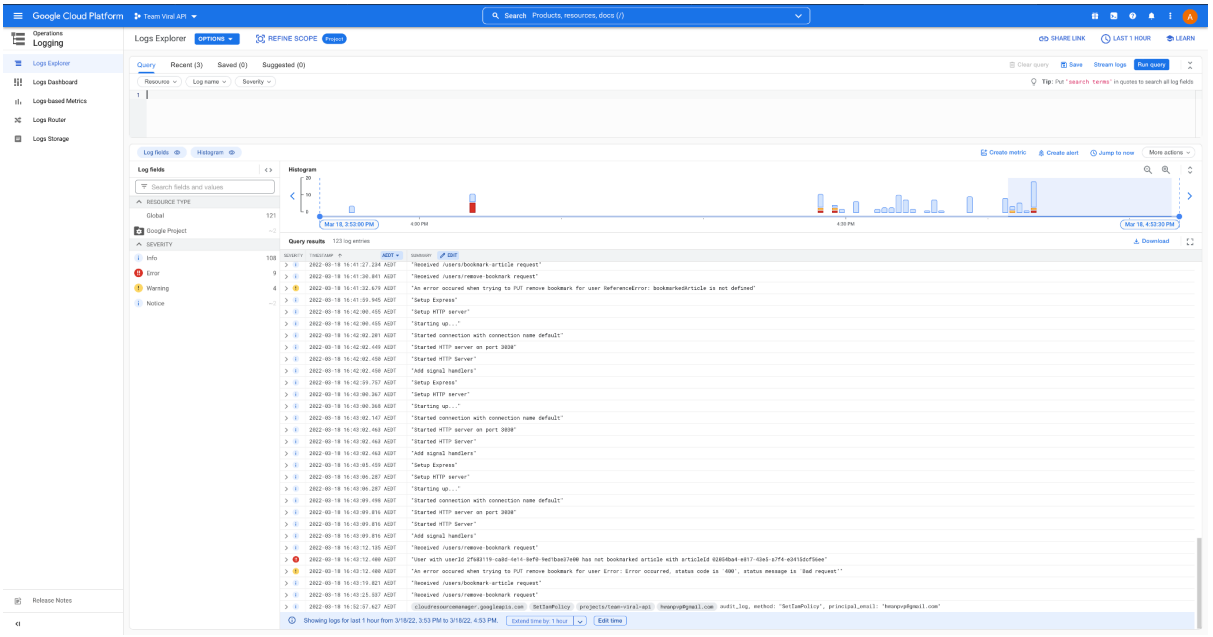
Widget

widget_id	**uuid, primary key**
dashboard_id	uuid, foreign key, non-nullable
widget_type	<String>, non-nullable
article_id	uuid, foreign key, non-nullable

Advanced Logging Capabilities

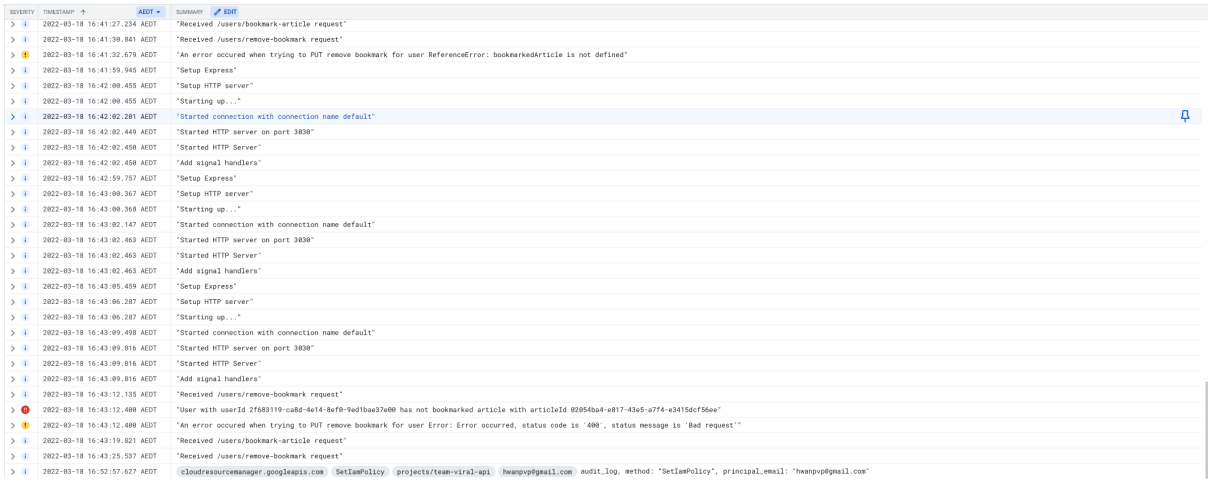
https://console.cloud.google.com/logs/

For logging capabilities, we have connected our logger in NodeJS to Google Cloud Platform to view all information and error logs and analytical data. All of our logs stream to the Google Cloud Platform, where we can collect, aggregate and categorise our data. This assists in tracking down errors and debugging where the problems are in a production environment.



Google Cloud Platform also provides useful metrics and tools to create graphs and charts based on the logs that are streamed. For example, we have a timeline of the warnings, errors and information logs that streamed across 24 hours:

We also have the complete collection of logs listed with their timestamps:



Challenges in Implementation

During implementation of our team's proposed tech stack for deliverable 2, major challenges and setbacks were discovered that either introduced a delay in resolving the issue or impacted the initial design of our project in the form of necessary amendments and compromises. The following section describes these issues in detail, and what steps were taken to resolve each of them. These issues are separated into categories based on where they were discovered in the tech stack.

Frontend

We have yet to begin frontend development and so have not faced any challenges thus far.

Backend

Unexpected Learning Curve with ORM

Querying a database from a server typically requires an ORM to establish a connection to the database and simplify results from SQL queries into interactable objects. In our case, the library that was used for this project is TypeORM. What our team did not anticipate was the learning curve that came alongside the addition of ORMs which included learning a library necessary to build SQL queries from Javascript functions. This brought in some difficulty as the SQL statements would need to be constructed, and then correctly translated into Javascript. ORMs can also be easily abused by team members unfamiliar with the database where they split what could have been one query into multiple simple queries as a way to avoid learning the ORM's database-specific functions. This introduces a massive hit to the performance and must be avoided as much as possible.

To avoid further complications on this matter, quality checks were performed for each route, checked to ensure that one route performs only one query, unless that route is complex enough to justify the usage of more than one.

Justification On Why We Avoid 405 Error

A design decision we have chosen with our backend server is that if a user hits a route with the incorrect HTTP method, they will get a 404 instead of a 405. This decision was made for a few reasons:

- Intercepting route requests to catch this type of user mistake causes complexity in codebase
- Introduce lots of redundancy and lots of boiler code with our Express framework.
- 404 error is sufficient and clear enough for this type of user error.

Our team has researched extensively to find a nice and efficient way for us to present this error. The solution we found was that each of our API controllers would accept all method types, and then we would have conditional statements to catch each HTTP method,

responding 405 for the invalid ones, and the service logic with response for the valid routes. Although with this we lose the readability and modularity of our code, which as a team we value more than the status code. Our final decision was that even this solution wasn't justified and worth the costs listed above.

Web Scraping

Extracting locations and dates

Scraping a website is bound to introduce several inconsistencies in the database. This is especially true for information that must be extracted from text written naturally inherent to the nature of the English language.

The first issue we had to resolve was how we would extract key information necessary for a report. Originally, to resolve the issue of recognising a location, the python library geograpy3 was used which uses NLTK to process natural language. However, upon inspection of a report's output, we found that its results were generally inaccurate with terms such as 'WHO' and 'Health' being a regular inclusion to our results amongst other locations that were clearly not cities, regions, or countries. This prompted us to reimplement our solution with another python library locationtagger, which gave more accurate results although deployment of this new solution will need to be delayed to the next deliverable.

Similarly, we used another library built on NLTK to identify dates called dateparser to interpret dates out of text. While the library does successfully scrape regular dates out of text, we encountered a similar problem to contextually finding locations where keywords such as 'on' and 'to' were recognised as individual dates, resulting in inaccurate event_dates being recorded. To resolve this issue, our team plans to modify the library's parameters to be more strict in the next deliverable.

Incorrectly Named Key Terms

A discovery our team made when creating the scraping algorithm was the naming mismatch of multiple key terms. Examples of these are "poliovirus", and any influenza variant. Since these critical problems were discovered near the end of Deliverable 2, we concluded that we would delay resolving these issues until the start of Deliverable 3. Our likely course of action after consulting with our mentor is renaming the terms so it better fits our data source.

Recognising When To Create a Report

A key part of this project is processing an article and deriving multiple report objects from the article's text. Our team found difficulty in differentiating when a report should be separated from another report.

In the first iteration of the algorithm, our team kept the algorithm simple, where only one report was generated from one article. This created a minimal database that allowed members of our team to implement the server to get started. However, more work would need to be done to achieve higher accuracy.

Our first attempt at splitting an article was to generate a new report based on when a syndrome or disease was found in the text. The algorithm could then try to look ahead and behind a certain number of words and try to identify key information from its surroundings. However, upon reinspection of the specification, the team recognised that a report could potentially hold multiple key terms, while only one date was allowed. This prompted the team to revise the algorithm and seek support from our mentor.

After a discussion with our mentor, our team came up with a revised version of the algorithm where a report is generated based on each paragraph in the article. If the report is incomplete (both diseases and syndromes array was empty, location array was empty, or event_date was empty), then the report was discarded. While successful on many articles, it also gave way to new issues mentioned below.

Preventing a 1 to 0..n Relationship Between Articles and Reports

While the new algorithm did successfully extract multiple reports from an article, there were also cases where no reports were generated. This posed a huge fundamental issue for our algorithm as it would prevent some articles from ever showing up in a search query.

To resolve this, we modified our scraper algorithm to have a fallback to generate a simpler report that did not separate the article's main text into multiple reports. While this did generalise the report drastically and potentially lead to inaccurate results, we considered it a worthwhile compromise over not showing the article.

Database

Limitation in Storage

While the deployment of our PostgreSQL server on Heroku went smoothly initially, the team did not foresee the limitation in the maximum number of records our server could hold at one given time. This first came to the attention of our team when one of the iterations of our scraper's report generation algorithm brought the count to over 10,000. A read-only restriction was imposed upon the database that prevented us from adding new records and updating existing records. In addition, this would prevent us from expanding our application and adding more functionality to our frontend.

As a workaround, our team rewrote the scraper's algorithm to be more strict. This reduces the number of potentially inaccurate reports, while also increasing the chance for the algorithm to unsuccessfully scrape an article. Beyond that, our team also plans to research other deployment options outside of Heroku that may allow us to scale our database beyond the 10,000 record limit.

Deployment

Deploying multiple applications in a Mono-repository

A notable restriction that exists in normal Heroku deployment is the one application per repository rule. As our entire project must be located in one repository as per specification, this introduced a problem in deploying all our necessary applications. Originally, we believed that it would be possible to deploy each application in separate branches. However, upon testing this, we deduced that Heroku recognises multiple Procfiles despite only having pushed one branch, causing a failure during deployment.

To resolve this issue, we would be using a community made buildpack ‘heroku-buildpack-multi-subdir’. This would allow us to call different Procfiles in a single repository for each part of our stack.

Conclusion

In conclusion, we have compared and assessed a number of technologies and tools to ensure our tech stack design is optimal for our project and aligns with our team members’ experience. Having outlined the core aspects of our project’s design here, this report will be a vital resource throughout the development of our project, reflecting on our choices and ensuring a cohesive implementation overall.