SENG3011 Design Details
Bobby Tables

Frontend

Framework

From its initial conception, it was decided that the frontend would be built with a **Javascript framework** that most members had some experience with. We decided to employ a framework over **vanilla Javascript** as we knew from the spec that we would be dealing with a **complex UI**, a task which frameworks could achieve better results in. With some brainstorming, the team came up with a shortlist of frameworks to achieve our goal:

- React
- Vue
- Angular
- Express
- Next.js

After some discussion and research, **Next.js,** a **framework** based on **React** was chosen to develop the frontend.

This decision was predominantly made due to the **higher levels of experience** with the framework, and also supporting technologies such as **React-router** (routing) and **Storybook** (UI) which will greatly assist in frontend development. It enables **React based functionality** such as **server-side rendering** and **static website generation**. This framework was chosen over React as although being very similar, Next.js was developed by **Vercel** which is our chosen deployment platform (discussed later).

That being said, **Vue**, **Angular** and **Express** are quite versatile frameworks and have no particular downsides apart from some **complicated syntax** that might arise from trying to learn them in a short period.

It was also decided that we would use **Typescript** in **conjunction** with **Next** as it offers benefits such as being a **compiled language** with **strong typing**. **Compilation errors** can be fixed **pre-compilation**, while also **executing code faster post compilation**.

Data Visualisation

Dealing with the visualisation of large datasets, we realised early on that we would require an efficient way to represent our data. From a multitude of libraries we shortlisted **Visx**, **Reaviz,** and **Chart.js**.

Visx is heavily based on **React** and as such will be **simple** to learn and employ in the frontend. As it is split into **multiple packages**, we have the ability to pick and choose various packages to **stop bloating** overhead dependencies. Compared to other libraries, it features many more niche and **unique graph types** allowing the styling of the data to be more **flexible**.
Similarly, Reaviz is another **modular** chart component library that employs the React framework. Although it offers less primitives than libraries such as Visx, Reaviz has **in depth documentation** allowing for **extreme versatility** in graph customisation.

Chart.js offers a minimalist yet clear representation of otherwise complex data, sporting **comprehensive documentation** and **animation support** for some extra edge.

We decided to use **visx** due to the **versatility** of the **graph primitives** it offered, combined with the **simplistic** yet **comprehensive documentation**, however we reserve the decision to complement the more elegant **visx** graphs with more simplistic **Chart.js** designs.

API

Similarly, we perceived the possibility of requiring an interactive map to aid in data visualisation. Map based graphs can only display so much detail, so we looked at utilising map APIs such as **Mapbox**, **OpenStreetMaps** and **Google Maps**. **Google Maps** is perhaps the most widely known mapping technology and features **excellent support** and **quality of data**, however since it is not open source, it incurs some monetary cost to use. **Mapbox** utilises the open source data from **OpenStreetMaps** and as such lacks the overall quality of data points compared to **Google Maps**, however increasing amounts of data have been added in recent years. It offers an **extremely versatile** set of maps and data representations through its API, with under 1000 requests per month being free. The only downside to an otherwise obvious choice is the **complexity** of the API and **steep learning curve** with the short turnaround time to develop the front end. As such, we will be proceeding with **MapBox** as our primary mapping API.

Styling

HTML and CSS will be used in **conjunction** with **React components** to provide further stylisation of the product. We will also employ various CSS libraries such as **Bootstrap**, **Material UI** and **Emotion** as we see fit. These component libraries are useful as they enforce the use of props over classes allowing for a distinction between the component and any CSS or HTML stylisations.

Deployment

For the deployment of the front end, there are endless platforms for doing so, each with their own strengths and shortcomings. Some of the more prominent options we reviewed included:

- Vercel
- Heroku
- Netlify

Vercel is a platform from which to deploy frontend frameworks. It features integrated React support alongside **flexible data fetching** and **inbuilt CI/CD systems**. Vercel also boasts **infinite scalability** which is always good for future proofing any upscales the product might encounter. Overall, it seems very powerful and offers a large magnitude of deployment support.

Heroku is one of the most popular deployment platforms out there being “built for developers, by developers”. It was designed to bypass many deployment overheads and thus features **easy deployment** and **excellent monitoring** and **scalability** for apps. Unfortunately, to use it to the full extent, there are paywalls which restrict its flexibility somewhat.

Netlify is also another popular hosting choice, built around **low latency**, **reliability** and **security**. Like most other platforms, it features **easy deployability** and supports **continuous deployment**. An extra feature is that it has free **SSL support**, however like **Heroku**, the pricing increases dramatically with the scale of the app. The

free subscription only allows for 1 concurrent build, and even then charges can be incurred when various limits are reached, such as bandwidth or build minutes.

As such, we decided to use **Vercel** due to its **React support**, as well as being recommended due to its ease of deployment.

Backend

Language

For our backend language, we have identified a couple of options; PHP, Python, Javascript (NodeJS). From these, we've chosen Python with the primary reason behind this being the familiarity of the language with our team. Whilst Python is not known for its good performance, its straightforward syntax, and flexibility will allow rapid development and prototyping, allowing us to deliver the project quickly, and easily provide maintenance once deployed. **Its availability of libraries and frameworks also add to its usability**. These other options are all good choices:

- PHP is easy to use and is commonly used in web development, however without a strict structure, projects can become difficult to maintain. Additionally, our team isn't familiar with the language.
- Choosing Javascript as our backend would give us a Javascript full-stack, which can increase team efficiency due to a better understanding of source code within the team as well as code reuse. It also has a relatively high performance and speed.

However, **Python** is the best choice for our team.

Framework

With Python decided as our language, possible web frameworks to use include:

- Flask
- Django
- FastAPI

Flask is the second most popular Python framework, taking a lightweight and modular design to help build a solid web application foundation. With its popularity, a variety of packages have been written to extend its functionality. Its flexibility makes it a good choice for our project.

Django is the most popular Python framework, and is a full-stack web framework however with the help of the package *Django REST framework*, could be utilised as only a REST API backend. Django comes built in with authentication as well as its own ORM that can be used with a variety of databases. While Django contains a lot of features that we would be able to use, it has an equal number that we don't need, and is therefore not the most suitable framework for us.

FastAPI is a high performance web framework similar to Flask. With full **asynchronous support**, it has been designed for speed and is on par with NodeJS and Go which helps to alleviate the **slower performance** gained by choosing Python. It is fast to develop, and is fully compatible with **OpenAPI** allowing us to spend more time

writing code rather than manually creating **Swagger documentation**. Because it is a newer framework, it has a smaller community, which means there aren't many educational resources and add-on packages available.

Another option is **AWS Lambda** which is more of a deployment option rather than a web framework. Code is organised into **Lambda functions**, which are run only when needed. A large advantage of this is that deployment is taken care of, allowing the team to focus solely on the code. It would also integrate with other **AWS instances** for **frontend** and **database deployment**, however an issue here is that most of the team is unfamiliar with this option, meaning time that could be spent on development must instead be spent learning.

We will be utilising **FastAPI** due to its performance, ease of use, and automatic documentation generation.

Testing

Possible testing libraries have been narrowed down to

- PyTest
- PyUnit
- Testify

PyTest is one of the more popular testing frameworks used by Python developers, meaning it has a sizable community and therefore a variety of plugins used to extend its functionality. It supports **unit testing**, **functional testing** and **API testing**, in **simple** and **concise** test suites. One downside of Pytest is that should we choose to move to a different library after writing some test cases, the work already completed will not carry over resulting in **duplicate work** being performed.

PyUnit is a built-in part of the standard python library meaning **no additional libraries** need to be installed, which helps maintain a clear list of dependencies. It allows clear and flexible test cases to be created, however does require a large amount of **boilerplate code**.

Testify can be used for unit, integration and system testing. It was designed to replace **PyUnit** and as such has similiar, as well as expanded upon, features. However, there is not a large amount of documentation available meaning it would be difficult for our team to learn this new framework. In addition to this, the development team of this library has suggested moving to **PyTest**.

From these choices, we will utilise **PyTest**. All members of our team have **previous experience** with this library, and with proper design we can overcome the con of this package by avoiding switching to a different testing framework.

Scraping

For our web scraper, there are two options; using requests alongside **BeautifulSoup** to download and parse web pages or using a web scraping framework. Our primary candidates for scraping frameworks are **Scrapy** and pyspider.

Pyspider is a less popular web crawler meaning there will be **less resources** online available for it, however it is **simple** and **easy to learn** which alleviates that downside. On the other hand, **Scrapy** is the most popular web

crawling framework with **extensive documentation** and resources available which will help combat its slightly harder learning curve. It is also **asynchronous**, meaning it performs better than Pyspider.

As suggested by the Spec, we will be using **Scrapy**. Being designed for the task, it will be best suited to accomplish our requirements rather than manually parsing HTML with **BeautifulSoup**. However, if there are any instances where we need to parse html, we will use **BeautifulSoup** to assist with this.

Deployment

For backend deployment, we have narrowed down three options: **AWS EC2** running **NGINX**, **Heroku** and **Deta.** AWS EC2 has the additional overhead of running a server and webserver, whilst Heroku and Deta are both **serverless** solutions, making them more suited to our needs. Deta holds the advantage of having a better free tier when compared to Heroku as it has no paid tier, and is also recommended by the developers of FastAPI. It is, however, a newcomer to the scene and as such lacks the community and associated resources of Heroku. Due to this, we will be using **Heroku** to host our backend.

For deployment of our scraper, we've identified **Heroku**, **Zyte**, and **AWS EC2** as possible solutions. All three have free tiers available, however **Zyte** is the best fit for us as it is hosted by the developers of **Scrapy**, the framework we are using, meaning it is highly coupled with their product. This also allows easier deployment and **CI/CD integration** as it **bypasses Scrapyd**, a program we would otherwise have to run in order to host the spider.

As such, our team will be deploying our web scraper using **Zyte Scrapy Cloud**.

Database

To choose a database, our team first had to decide on if we wanted a **relational** or **non-relational** database. The relationship between articles and reports in the spec do point towards a relational database, though on the other hand choosing a non-relational database would allow us to store a large amount of data that is less structured, and would also allow faster application development without needing to worry about database schema (which could need to change at any point due to the nature of Agile product development) . This decision to use a **non-relational database** is also helped by the fact that non-relational databases scale horizontally, meaning it is both **easier to scale** through adding more machines, and is **more reliable** as data is split and replicated across nodes.

Our options for non-relational databases were therefore **AWS DynamoDB** and **MongoDB** after eliminating **SQLite** and **PostgreSQL** from our preferred choices. DynamoDB is a managed service, whereas MongoDB is only a software, however can be hosted as a managed service on MongoDB Atlas. Whilst DynamoDB is a strong choice, it is even stronger when part of an AWS ecosystem, however choosing it does **vendor lock** you into AWS whilst MongoDB allows you run a **self-managed cluster** or move to a different provider. MongoDB has more flexible querying when compared to DynamoDB's key-value queries, as well as better **JSON support** - something essential for our project where we will be serving JSON through our backend API.

Overall, **MongoDB** will be our chosen database and will be deployed through **MongoDB Atlas**.

Challenges and Shortcomings

Algorithmic

The parsing of our scraper is designed so that from the data scraped, a "**window**" is then used to analyse small portions of the larger article. This "window" trawls through the data and attempts to extract **keywords** or **phrases** and match them to predefined lists of **locations** and **diseases**. These are then formatted into a **report** and stored in our **database**.

Locations

Extracting **location names** proved to be challenging as, unlike dates, their only distinguishing feature is being capitalised (most of the time). Our first thought was to simply have a **list of locations** that we could directly compare words within the window to. We utilised an **online JSON** which contained location data, including **country**, **state** and **city/town** names. This was then **parsed** into a more readable format for our uses.

Unsurprisingly this fell short due the fact that many locations are often multiple words, and would return **partial or incomplete matches**. The next logical step was to use natural language processing. We chose to use the **Google Cloud NLP Api**, as once set up, we could send each window and receive a response which would classify various words as locations. This new list proved to be **more refined**, however we were still plagued by partial matches and incorrectly identified locations. For example, phrases such as "**Central Africa**" would return results for both "**Central**" and "**Central Africa**", obviously, only one being correct. Luckily, in the response, the API includes a **wikipedia article link** in the **metadata** if one exists for proper noun locations. Location based words like "**constituency**" do not include such links, and as such we could further **filter phrases**. Once again, this **refined list** was compared to our **location JSON** and any **duplicate mentions** were removed per report.

Dates

Date strings were both simultaneously easier and harder to identify. To help us extract dates, we employed the **datefinder library**. Using the **datefinder.find_dates** function, **complete dates** (17 November 2020), were easy to parse and convert to the correct format. However, **incomplete dates**, or dates that were preceded by a year, which would make sense grammatically but not programmatically were much harder to deal with. If a day and month were missing, it would **default** to the **current date**, and similarly, if a year was not found, it would **default** to **2022**. The function had a **boolean parameter** "**strict**" which would only match complete dates when true. Although this would be ideal, many of the dates in the date were not represented in this format, and would not be picked up. **Date ranges** were also troublesome as it was sometimes hard to distinguish if the dates belonged to **different reports** in the same window.

Disease/Syndromes

Some diseases and syndromes were hard to detect, while others were quite straight forward. **Single word phrases** such as "Monkeypox" could easily be found in either the disease JSON or syndromes JSON. **Multiple word phrases** such as "acute fever and rash" had a larger chance of returning **false positives** as they would only **partially match** or not contain the full syndrome string. In some cases, viruses such as "**influenza a/h5n1**" would be referred to as **H5N1** which just required some **further manipulation** to match.

The sample list of diseases and syndromes were quite **primitive** and we quickly found that not all the reports could be processed off them, so we **expanded** the lists through some research into common outbreak illnesses.

Documentation

Due to the **unfamiliarity with FastAPI** and its auto-generated Swagger page, development of the documentation was often tedious and slow. Multiple elements of the **documentation could be improved** and made clearer but could not be done **due to limited understanding and dwindling project time**. Examples of the struggles encountered include being unable to show multiple example response values, the inability to delete auto-generated responses and failure in removing unnecessary schemas.

Logging

The logging system was developed as an additional functionality for the API and was done through the use of the **built-in Python logger** library. The reasoning being that its **simplicity allowed for a quick development** which was pivotal due to the **limited project time remaining**. The current logging system exists as a **Python decorator** which wraps around route functions making it easy to extend to additional backend routes. It is currently designed to save information only when an error is encountered server side. The data being saved includes the current time, stack trace, request method, url and parameters with an example output shown below.

```
03/20/2022 06:14:56 PM   File "./helpers.py", line 102, in wrapper
   return await func(*args, **kwargs)
 File "./main.py", line 155, in article_content
   return {"content": helpers.get_article_section(articleId, "Content")}
 File "./helpers.py", line 86, in get_article_section
   article = get_article_dict(article_id)
 File "./helpers.py", line 93, in get_article_dict
   raise HTTPException(status_code=404, detail={"error_message": "No article found with
that given id"})

03/20/2022 06:14:56 PM GET
03/20/2022 06:14:56 PM http://127.0.0.1:8000/article/9019/content
03/20/2022 06:14:56 PM The parameters that triggered this error
03/20/2022 06:14:56 PM {'articleId': 9019, 'version': 'v1.0', 'request':
<starlette.requests.Request object at 0x10aad7be0>}
```

Security

There have been overall **issues with securing the username and password** required to connect to the database. This is due to the fact that our current deployment **requires them to be hard coded** in as strings for Github to automatically deploy. While this is evidently not secure, **time constraints** meant that no alternatives could be implemented in time for the deliverable, as such it was left as a **temporary solution**. Luckily the credentials are easy to regenerate and the repository being private should reduce the possibility of security issues arising, but a more secure and permanent solution will be needed for the next deliverable.