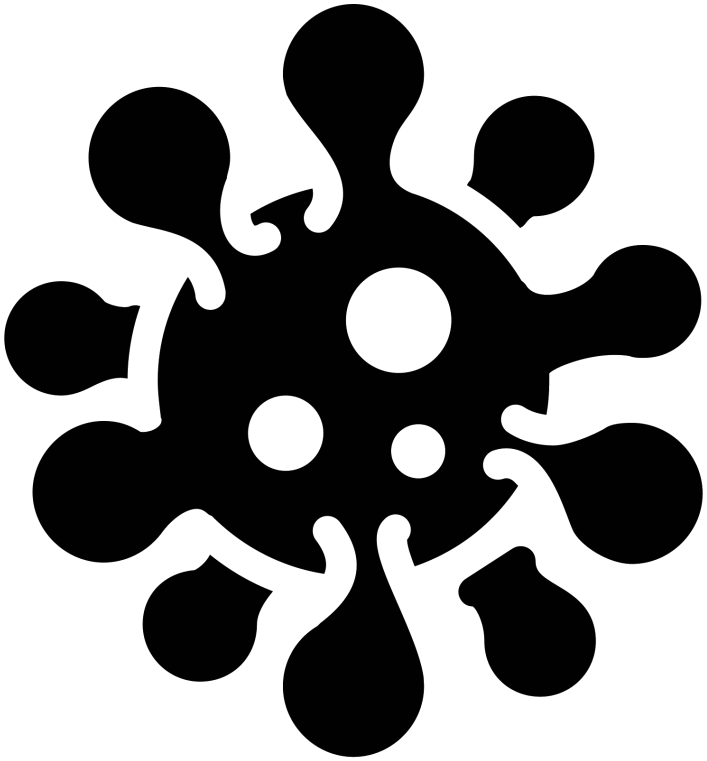
Design Details Report



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 will be built. The team has varying levels of understanding as to how the API will work 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 1. 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. We have identified that integration will likely be a source of trouble and hence this document (in particular the software architecture diagram) will help us identify and plan for any issues associated with integrating everyone's assigned parts.

As of writing this report, the specifications for deliverables 2 and 3 have just been released and as such the team has little understanding of how they will impact our design. Therefore, the team notes that any aspect of our design may change to address the requirements laid out by these later deliverables. If such a change is necessary, the team will revise our current design and modify it where necessary. To provide some guidance on what these modifications should be like, the team has agreed that their impact should be minimal in order to reduce disruption to progress as we have identified that transitioning from one design to another will require time (e.g. regression testing will be required, extra research into new components we'll need to use).

This report is divided into three sections that touch on a particular aspect of our design. 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
API Design	An overview of how we intend to develop our API and how we will enable developers to use it in web service mode.	To communicate an early understanding of how our API will work.
Designing API Endpoints	A discussion of how parameters will be passed into our API and how our API will respond.	To communicate an early understanding of how our API will be used.
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.

API Design

Good principles of API Design

In designing our API service, we first needed to define what good API design entailed through research. First, we looked at the origin and definition of HTTP requests outlined by The Internet Society's document *"Hypertext Transfer Protocol -- HTTP/1.1"*. Here we developed an understanding of the main operations and ways to implement our API service through GET, POST, PUT and DELETE methods. We further sought out the best ways to implement these methods through an article by Swagger *"Best Practices in API Design"* , where we identified our 3 main goals for our RESTful service:

- Making it easy to work with
- Hard to misuse
- Concise whilst also being complete

This research has provided us with the background and an in depth understanding of what exactly we are developing and designing, whilst also guiding us in the ways to practice good design.

From this research, we began designing our API service, highlighting the routes, parameters and responses as examined in the next section.

Parameters

For our POST, PUT and DELETE operations we will pass parameters to our APIs through the path, query string or request body. The path will be utilised for unique identifiable information such as IDs for particular articles or reports. Query strings will be used for small instances or information which won't introduce complexity or clutter. Request bodies will be the main transport for large payloads of information where the parameter might include objects, which would otherwise clutter the API path/route or query strings.

For GET operations the aim is to keep additional queries to a minimum, however as per the specification, we have some crucial pieces of information which need to be passed as a criteria to filter data on articles and reports. It is considered bad practice, as outlined in the *"Hypertext Transfer Protocol -- HTTP/1.1"* document, to have a request body with GET requests, and in our case we need to pass objects and large payloads to our backend with a GET API request. Our solution is to utilise the headers to contain this information, ensuring that we maintain our design to the standards outlined in the definition for HTTP requests and other design guides such as the Swagger article.

Versioning

We acknowledge that versioning is an important aspect of our API design as other teams will be utilising our API, as a result we intend to version our API via the prefix in the route. For our initial implementation we intend to have our routes / URL structured to include the API version, e.g. *{URL}/v1/articles*. This is a common API design pattern that is followed and highlighted in the *"Best Practices - API Design"* report by Microsoft. With this, we can

maintain older routes as well as update them according to how our requirements might change for our frontend application design.

Responses

Our API's will follow a simple response structure for users to expect, differing from error and successful responses.

The object interface for success responses will look as follows (JavaScript syntax):

```
{
  status: string "Success";
  payload: object (e.g. articles);
}
```

The object interface for error responses will look as follows (JavaScript syntax):

```
{
  errorCode: number (e.g. 400);
  errorMessage: string (e.g. Failed to retrieve reports);
}
```

We define our status codes in the following table:

Status code	Definition	Circumstances (examples)
200	Success response	• Articles successfully retrieved from database • Reports successfully retrieved from database • User is successfully authenticated
400	Bad request (didn't pass middleware, authentication, or wrong parameters)	• Missing fields in request body of article search request • User isn't authenticated to make request
404	Non-existent route	• User tries to access route which doesn't exist as an endpoint
500	Internal server error (Something went wrong with backend)	• Database connection in backend is down • Downstream systems are down • Edge case not covered in backend logic

By these definitions, users of the API will have a certain structure of what's expected from each route, and will be able to get further information from our Swagger file.

Designing API Endpoints

Querying

According to the specification, our API must incorporate one endpoint used to filter out irrelevant articles to our interests and must accept these parameters in some form:

- Period of interest;
- Key terms;
- and Location.

Route name:

```
/search
```

Since this request only reads the data, this route must be a GET request. As mentioned above in **Parameters,** we also aim to avoid cumbersome URL lengths through long query strings. Hence, these parameters will be passed through a custom header with the respective names and variable types:

```
{
	period_of_interest : {
		start: <String:date>,
		end: <String:date>
	},
	location: <String>,
	key_terms: [<String>]
}
```

As previously mentioned in **Responses**, the endpoint must return some form of payload as a response. For this endpoint, the specification requires a list of article objects to be returned in some form through the payload. Based on Appendix C of the specification, the response form for a single article object will be represented by:

```
{
	article_id: <String:UUID>,
	url: <String>,
	date_of_publication: <String:date>,
	headline: <String>,
	main_text: <String>,
	reports: [<Report>]
}
```

Whereas a report article object will be represented as:

```
{
      report_id: <String:UUID>,
      diseases: [<String>],
      syndromes: [<String>],
      event_date: <String:date>,
      locations: [<String>]
}
```

An example response of a successful filter can be found below.

Method	**Response**
GET	{ "status": "Success", "payload": [{ "article_id": "cdd7131b-2b76-40d0-a9b2-db8dca233f98" "url": "https://www.who.int/emergencies/disease-outbreak-news/item/circulating-vaccine-derived-poliovirus-type-2-(cvdpv2)-yemen", "date_of_publication": "9 December 2021", "headline": "circulating-vaccine-derived-poliovirus-type-2-(cvdpv2)-yemen", "main_text": "...", /*Excluded*/ "reports": [{ "report_id" : "e2d1f78f-7905-4ddb-aacf-9b31a2e71b01" "diseases": ["polio"], "syndromes": ["irregular fever"], "eventDate": "2021-12-09T00:00:00", "locations": ["Yemen"] }] }] }
Request Route	
/search	
Headers	
{ period_of_interest : { start:"2021-12-09T00:00:00", end: "2021-12-09T00:00:00" }, location: "Yemen", key_terms: ["Virus"] }	

Outside of the necessary search route, our team has also added additional routes that will likely be used by our team and other teams in the future.

Pagination

An endpoint that would separate our articles into more manageable chunks was deemed necessary to browse through our information.

Route name:

```
/articles
```

Pagination of our database will be offered through this route allowing for API users to paginate through our data, through the parameters `offset` and `limit`. Since these parameters are sufficiently short, they will be accessible through the URL's query.

An example of a successful query for the 15th page where only 3 articles are shown per page looks like this:

Method	**Response**

GET	<pre>{
	  "status": "Success",
	  "payload": [
	  {
	  "article_id": "6ddaa7cb-074e-451f-882c-ef61947042f6"
	  "url":
	"https://www.who.int/emergencies/disease-outbreak-news/item/cholera-cameroon",
	  "date_of_publication": "16 December 2021",
	  "headline": "Cholera - Cameroon",
	  "main_text": "...", /*Excluded*/
	  "reports": [
	  {
	  "report_id" : "11efc994-3421-43c2-8d1e-cd13d7ee63c2"
	  "diseases": [
	  "cholera"
	  ],
	  "syndromes": [
	  "death"
	  ],
	  "eventDate": "2021-12-16T00:00:00",
	  "locations": ["Cameroon"]
	  }
	  ]
	  }, {
	  "article_id": "cdd7131b-2b76-40d0-a9b2-db8dca233f98"
	  "url":
	"https://www.who.int/emergencies/disease-outbreak-news/item/circulating-vaccine-derived-poliovirus-type-2-(cvdpv2)-yemen",
	  "date_of_publication": "9 December 2021",
	  "headline":
	"circulating-vaccine-derived-poliovirus-type-2-(cvdpv2)-yemen",
	  "main_text": "...", /*Excluded*/
	  "reports": [
	  {
	  "report_id" : "e2d1f78f-7905-4ddb-aacf-9b31a2e71b01"</pre>

Request Route	
/articles?limit=3&offset=15	
Headers	
{}	

<table>
<tr><td></td><td><pre>
          "diseases": [
            "polio"
          ],
          "syndromes": [
            "irregular fever"
          ],
          "eventDate": "2021-12-09T00:00:00",
          "locations": ["Yemen"]
        }
      ]
    }, {
      "article_id": "a7062eac-1238-41c8-8d11-f71b58211faa"
      "url":
"https://www.who.int/emergencies/disease-outbreak-news/item/yellow-fever---gha
na",
      "date_of_publication": "1 December 2021",
      "headline": "Yellow Fever - Ghana",
      "main_text": "...", /*Excluded*/
      "reports": [
        {
          "report_id" : "2053c15d-e4f9-43ed-811a-2d60bda2f2b0"
          "diseases": [
            "Yellow fever"
          ],
          "syndromes": [
            "irregular fever"
          ],
          "eventDate": "2021-12-01T00:00:00",
          "locations": ["Ghana"]
        }
      ]
    }
  ]
}
</pre></td></tr>
</table>

Suppose a negative number was passed into the limit. Instead of a successful response, the response would now display with an error message:

Method	**Response**

GET	<pre>{
	  errorCode: 400,
	  errorMessage: "Invalid limit. Limit must be greater than 0."
	}</pre>

Request Route	
/articles?limit=-1&offset=15	
Headers	
{}	

Selecting a single item

Users of our API may also specifically request information on one article or report. To address that issue, our team has included endpoints for both reports and articles that will take in an `id` parameter which will be accessible via the URL's path.

Route names:

```
/reports/{report_id}   /articles/{article_id}
```

Here is an example call that requests for an article with the id, 'cdd7131b-2b76-40d0-a9b2-db8dca233f98'.

Method	**Response**

GET	<pre>{
	  "status": "Success",
	  "payload": {
	  "article_id": "cdd7131b-2b76-40d0-a9b2-db8dca233f98"
	  "url": "https://www.who.int/emergencies/disease-outbreak-news/item/circulating-vaccine-derived-poliovirus-type-2-(cvdpv2)-yemen",
	  "date_of_publication": "9 December 2021",
	  "headline": "circulating-vaccine-derived-poliovirus-type-2-(cvdpv2)-yemen",
	  "main_text": "...", /*Excluded*/
	  "reports": [
	  {
	  "diseases": [
	  "polio"
	  ],
	  "syndromes": [
	  "irregular fever"
	  ],
	  "eventDate": "2021-12-09T00:00:00",
	  "locations": ["Yemen"]
	  }
	  ]
	  }
	}</pre>

Request Route	
/articles/cdd7131b-2b76-40d0-a9b2-db8dca233f98	
Headers	
{}	

Selecting all items

Unexpected behaviour will likely occur during our implementation process. Having an endpoint that will allow us to have an inside look on what is stored within our database, will be beneficial when debugging and allow us to identify whether faulty data is the cause of it. However, since the database will be storing large amounts of data, this route should be limited to debugging purposes.

Route names:

```
/reports/dump   /articles/dump
```

An example of what a successful dump of all articles looks like is below.

Method	**Response**

GET	<pre>{
	  "status": "Success",
	  "payload": [{
	  "article_id": "cdd7131b-2b76-40d0-a9b2-db8dca233f98"
	  "url": "https://www.who.int/emergencies/disease-outbreak-news/item/circulating-vaccine-derived-poliovirus-type-2-(cvdpv2)-yemen",
	  "date_of_publication": "9 December 2021",
	  "headline": "circulating-vaccine-derived-poliovirus-type-2-(cvdpv2)-yemen",
	  "main_text": "...", /*Excluded*/
	  "reports": [
	  {
	  "diseases": [
	  "polio"
	  ],
	  "syndromes": [
	  "irregular fever"
	  ],
	  "eventDate": "2021-12-09T00:00:00",
	  "locations": ["Yemen"]
	  }
	  ]
	  },
	  /* ... and over 2000 other articles*/
	  ]
	}</pre>

Request Route	
/articles/dump	
Headers	
{}	

Tech Stack

Software Architecture Diagram

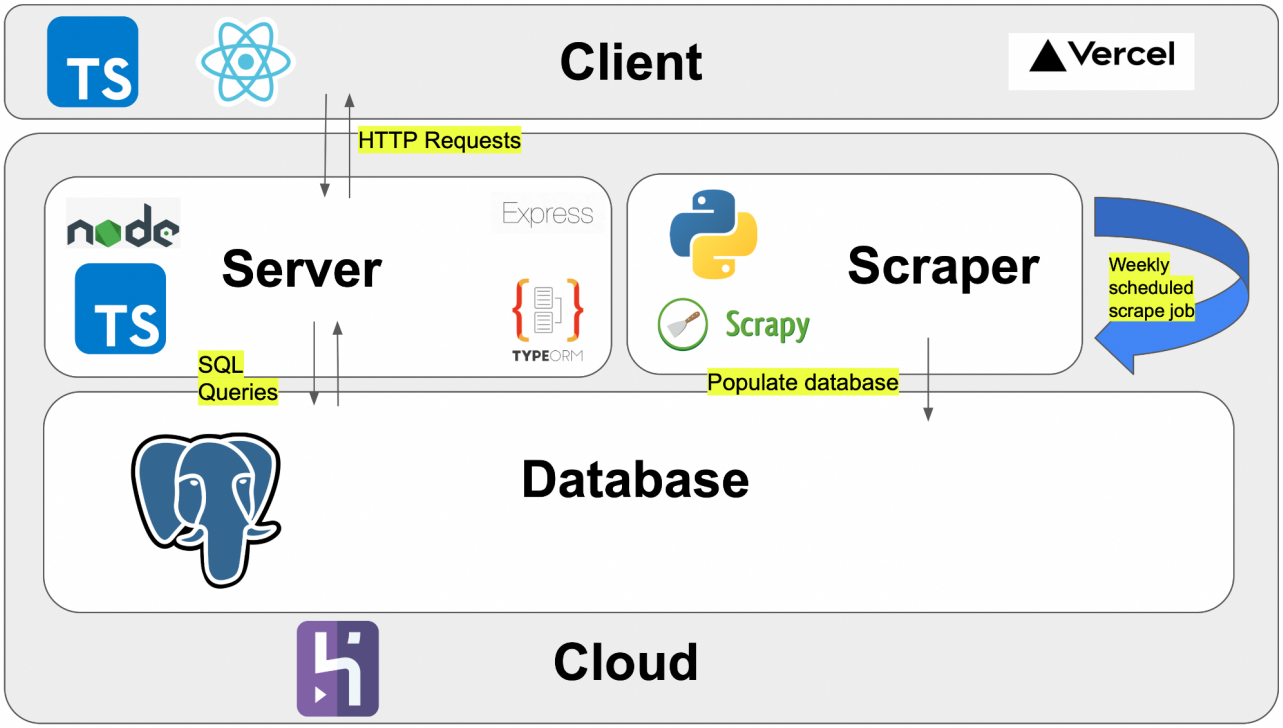


Figure 1. Visual diagram of the tech stack

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

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

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

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

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.