SENG3011 Design Details
Bobby Tables

API Design 3
API Module Design 3
API in Web service 4
Specifications 4
Versioning 4
Error handling 5
Query strings and request body 5
Route naming 6
API Routes 6
Documentation 6
Frontend 7
Framework 7
Data Visualisation 7
API 8
Styling 8
Deployment 8
Backend 9
Language 9
Framework 9
Testing 10
Scraping 11
Deployment 11
Database 12
Tech Stack Diagram 13
Appendix 14
Appendix A - Route Table 14

API Design

API Module Design

We have chosen to develop a **REST** (Representational State Transfer) API using **HTTP** as its application protocol. REST is a **resource-oriented** architecture style. When a client request is made to a server via the API, a representation of the resource is transferred to the client via HTTP. The representation will be delivered in **JSON** in our project because of its popularity and readability.

For our project, the API identifies **articles** and **reports** as resources from the main data source (WHO Disease Outbreak News). According to the project specification, an **article** is a URL on the website consisting of information about one or more cases of a disease. A **report** is one of the cases within an article.

Each of the resources will be uniquely identified by a uniform resource identifier (**URI**). For example, the URI for an article with an ID e.g. 1 could be

```
/articles/1
```

Clients/users request **representations of resources** from the server in JSON. For example, a GET request to the URI above could return a response following the format in the project specification

```
{
        “article_id”: <integer>,
        “url”: <string>,
        “date_of_publication”: <string::date>,
        “headline”: <string>,
        “main_text”: <string>,
        “reports”: [<object::report>]
}
```

Other HTTP methods including POST, PUT, DELETE, etc will be used as well. Sample request and response body for different API routes are provided in Appendix A.

The API module consists of the following parts: the **API**, a **scraper** for data gathering, and a **database** to store the data gathered from the source. The scraper is able to automatically access the WHO website at a given frequency or when the client requests the latest data. The extracted data will be stored in the database for permanent access.

The scraping framework we will use for the project is **Scrapy**. It **crawls and processes** articles from the data source and identifies different parts of an article e.g. epidemic reports, WHO response, assessment and advice, etc. The following is a brief of the process

- Define an **item** field based on the data we extract from the website, e.g. URL, publishing date, title (headline), etc.
- A **spider** can be initialised to crawl data.
- Data will be **stored** and **serialised** in JSON files, further in the **database** (MongoDB Atlas).

MongoDB is selected to be the **database** for the project. The JSON data from the scraper will be stored and accessed for various purposes of the API e.g. create new collections with new articles. The extracted data will be **indexed** based on client requirements e.g. title, date, etc for easier and more efficient searching.

API in Web service

To ensure that the API has the ability to run in Web

- The API is run using HTTP and processes data in JSON.
- The API is secured by an SSL certificate (HTTPS).
- The API is cloud-based as it is deployed on a cloud computing server (Heroku).

Specifications

Some fields are outlined by specification to follow specific formats, as such there are no possible alternatives that can be used. These fields are **url, date of publication, headlines, key terms, location, articles and report**s, which all follow the format outlined within the specification.

Versioning

Request headers will be used to keep track of the version of APIs being created. This will have a **default value of the earliest version** if a request is sent without a version being specified. Header versioning was selected as it is **cache friendly, relatively simple to implement and adaptable to changes**. Other alternatives were considered but not selected due to the reasons given below.

No versioning - While **simple to implement**, it would make the API **much more inflexible** as new updates would need to ensure that it does not alter or impact previous functionality. Since more features may need to be added in the future to support the front end, this method would be a poor choice.

URI versioning - Storing the versioning within the URI of the API would ensure the client is required to **explicitly state the version of the API** being used and performs similarly to header versioning. However it was not selected as it would **require new sets of endpoints to be established each time** a new version is released making it more tedious to navigate the backend from a development standpoint.

Error handling

Following standard convention, errors will return an appropriate HTTP status code alongside a short message detailing what went wrong in the response body.

Error code	Circumstance	Examples
2xx	Request is successful	200 Success - Successfully getting a webpage.
3xx	Informing the client on any changes in regards to the location of the resource specified by the url.	304 Not modified - Getting a webpage that has been cached and has not been modified since the cache.
4xx	The request contains an error.	400 Bad Request- Date given is not correctly formatted. 404 Not Found - Cannot find an article corresponding to the given URL. 405 Method Not Allowed - Calling a POST request when the route does not handle that method.
5xx	The server encountered an error that is preventing the request from being handled normally.	503 Service Unavailable - The API is undergoing maintenance.

Query strings and request body

Arguments to a route can typically be passed in through either the URL as a query string or through the request body directly. The convention used within this API is to have **parameters that would be used to filter the data**, such as offset, limit and start_date, **be placed within the query string**. **Parameters that impact the actual data returned**, such as email, **are placed within the request body**. This method was chosen as it allowed us to **separate parameters that are most likely to be consistent across requests from those more likely to be changed**, reducing the possibility of accidentally modifying parameters that should be constant. Since **according to HTTP convention, GET requests should not contain a request body**, the convention outlined above only applies to POST requests. An artifact of this method is long URLs for getting a list of articles. While this could result in a more tedious debugging process for the frontend, it can be mitigated by using existing Javascript functions that can parse and display the URL in a more readable format.

Route naming

Following the principles of REST being focused around resources, routes such as /article correspond to the resource of a collection of articles. As such GET, POST and DELETE requests correspond to performing the commands on the entire collection. Specified /article/{articleId}/ routes corresponds to an article with the given articleId. This route design ensures further routes, such as /article/{articleId}/content, **cannot be called without an articleId** present reducing the likelihood of missing required parameters.

API Routes

Documentation

To document the available API routes, Swagger UI and Stoplight were both applications that were considered. However, since **Swagger has a 2 week free trial while Stoplight is free**, the latter was chosen as it was better financially. The link to the relevant documentation is given below.

https://bobbytables.stoplight.io/docs/pandemic-api/YXBpOjQzMjI3NTU4-pandemic-api

Alternatively the route table detailing the parameters, errors and testing can be found in Appendix A.

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 it's **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, it's straightforward syntax, and flexibility will allow rapid development and prototyping, allowing us to deliver the project quickly, and easily provide maintenance once deployed. **It's 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. It's 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.

There are two further options for frameworks: **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 it being slightly harder to learn. 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**.

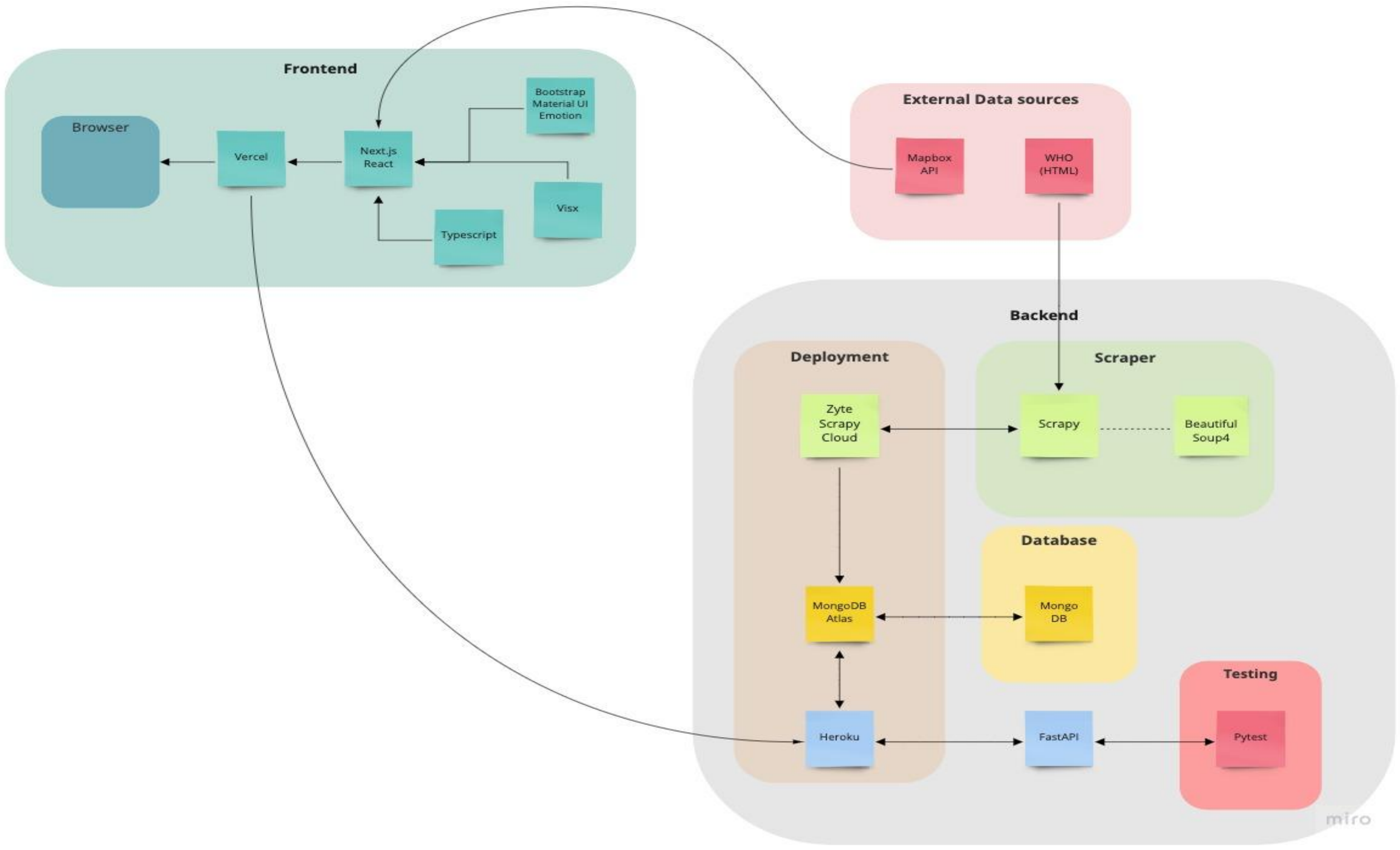
Database

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**.

Tech Stack Diagram



https://miro.com/app/board/uXjVOIbvG_w=

Appendix

Appendix A - Route Table

To keep the table brief, headers have been omitted but request headers follow the following format for all routes.

```
{
   “version”: “v1.0”
}
```

Error messages return a corresponding status code and a body with the following format.

```
{
   “response”: “Error message indicating what went wrong”
}
```

Route	Route Description
/article?limit=10&offset=0&start_date=2021-02-01T00:00:00&end_date=2021-02-01T00:00:00&location=Australia	**Request Type:** GET **Description:** Gets a list of articles from the data source according to the given parameters. **Parameters:** ● location - String of countries to extract articles from ● start_date - String of date in “yyyy-MM-ddTHH:mm:ss” format ● end_date - String of date in “yyyy-MM-ddTHH:mm:ss” format ● keyterms - String with comma separated keywords that needs to be present in an article for it to be returned ● offset - Integer in url representing the amount of headlines to skip, default is 0 ● limit - Integer representing maximum amount of results to return, default is 20, maximum is 50 **Response:** ● article - List of articles and their corresponding articleId for future route calls ● max_articles - Integer indicating the maximum number of results

	found with given parameters **Testing:** ● Ensuring articles returned contain key terms ● Ensuring articles returned are from the location ● Ensuring articles returned is published between specified dates ● Ensuring number of articles returned is lower than the limit ● Ensuring an error is raised if offset exceeds number of results available ● Ensuring an error is raised when date is incorrectly formatted ● Ensuring an error is raised when end_date is before start_date **Errors (Reason - Status code - Message):** ● Invalid offset number - 400 (Bad Request) - "Offset exceeds number of results available" ● end_date < start_date - 400 (Bad Request) - "End date is before start date" ● Date incorrectly formatted - 400 (Bad Request) - "Date incorrectly formatted" ● Any input missing - 400 (Bad Request) - "Parameter missing"

	Sample response body:
	
	`{`
	`   "articles": [{`
	`      "article": {`
	`         "url": "ww.sampleurl.com",`
	`         "date_of_publication": "2022-01-01T00:00:00",`
	`         "headline": "Ebola",`
	`         "main_text": "Sample main text",`
	`         "reports": []},`
	`      "articleId": 1}`
	`   ],`
	`   "max_articles": 1`
	`}`

/article/{articleId}/content	**Request Type:** GET **Description:** Gets a string of text indicating content for a given article. **Parameters:**

	● articleId - Integer representing the id for a article **Response:** ● content - String detailing the HTML content of an article **Testing:** ● Ensures response is associated to the correct article ● Ensures error is raised when called on page with invalid articleId **Errors:** ● Invalid articleId - 404 (Not Found) - “No article found with given id”

	Sample Response:
	
	`{`
	`  “content”: “Example content response”`
	`}`

/article/{articleId}/source	**Request Type:** GET **Description:** Gets link for the article for a given article **Parameters:** ● articleId - Integer representing the id for a article **Response:** ● link - String detailing the HTML content of an article **Testing:** ● Ensures response is associated to the correct article ● Ensures error is raised when called on page with invalid articleId **Errors:** ● Invalid articleId - 404 (Not Found) - “No article found with given id”

	Sample Response:
	
	`{`
	`  “link”: “abc.testing123.com”`
	`}`

/article/{articleId}/assessment	**Request Type:** GET **Description:** Gets the WHO assessment for a given article **Parameters:** ● articleId - Integer representing the id for a article **Response:** ● assessment - String detailing the assessment section for the given article **Testing:** ● Ensures response is associated to the correct article ● Ensures error is raised when called on page with invalid articleId **Errors:** ● Invalid articleId - 404 (Not Found) - “No article found with given id”

	Sample Response:
	
	`{`
	`  “assessment”: “WHO assessment”`
	`}`

/article/{articleId}/advice	**Request Type:** GET **Description:** Gets link for the article for a given article **Parameters:** ● articleId - Integer representing the id for a article **Response:** ● link - String detailing the HTML content of an article **Testing:** ● Ensures response is associated to the correct article ● Ensures error is raised when called on page with invalid articleId

	Errors: ● Invalid articleId - 404 (Not Found) - “No article found with given id”

	Sample Response:
	
	`{`
	`“advice”: “WHO advice”`
	`}`

/article/{articleId}/response	**Request Type:** GET **Description:** Gets the WHO response for a given article **Parameters:** ● articleId - Integer representing the id for a article **Response:** ● response - String detailing the WHO response of an article **Testing:** ● Ensures response is associated to the correct article ● Ensures error is raised when called on page with invalid articleId **Errors:** ● Invalid articleId - 404 (Not Found) - “No article found with given id”

	Sample Response:
	
	`{`
	`“response”: “WHO response”`
	`}`

/subscriber	**Request Type:** POST **Description:** Sets up an email to receive information whenever a new article is added for a list of given countries **Parameters:**

	● email - String representing email to send to ● location - List of strings representing countries to be notified on whenever a new article comes out **Response:** ● response - String indicating whether or not subscription was successful **Testing:** ● Ensure route correctly adds a valid email ● Ensures email is correctly notified when a report with the given country is added ● Ensures error is thrown if location parameter is empty ● Ensures error is thrown if email given is invalid **Errors:** ● Illegal email - 400 (Bad request) - Given email is invalid ● Empty location string - 400 (Bad request) - Location parameter is empty

	Sample Query:
	
	`{`
	`  "email": "testin123@gmail.com",`
	`  "location": ["Australia", "New Zealand"]`
	`}`
	
	Sample Response:
	
	`{`
	`  "response": "Success, email has been subscribed"`
	`}`

/subscriber	**Request Type:** DELETE **Description:** Sets up an email to receive information whenever a new article is added for a list of given countries **Parameters:** ● email - String representing email to remove from

	Response: ● response - String indicating whether or not the removal of the subscription was successful **Testing:** ● Ensure route removes a valid email ● Ensures error is thrown email given was not already subscribed ● Ensures error is thrown if email given is invalid **Errors:** ● Illegal email - (400) bad request - Given email is invalid ● Email not subscribed - (400) bad request - Given email is not subscribed

	Sample Query:
	
	`{`
	   `“email”: “testin123@gmail.com”,`
	`}`
	
	Sample Response:
	
	`{`
	   `“response”: “Success, email has unsubscribed”`
	`}`