

SENG3011 Deliverable 1
Bobby Tables

Design and Implement the API Module	4
Run the API in Web service mode	4
API Design	5
Specifications:	5
Versioning:	5
Error handling:	5
Query strings and request parameters:	5
Route naming:	6
API Routes	6
Frontend	7
Framework	7
Data Visualisation	7
API	7
Styling	8
Deployment	8
Backend	9
Language	9
Framework	9
Testing	10
Scraping	10
Deployment	11
Database	11
Project Management	12
Roles & Responsibilities	12
Project Plans	13
Software Tools	13
Appendix	14
Appendix A	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 reports**, 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/YXBpOjOzMjI3NTU4-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 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. **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. 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 similar, 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**.

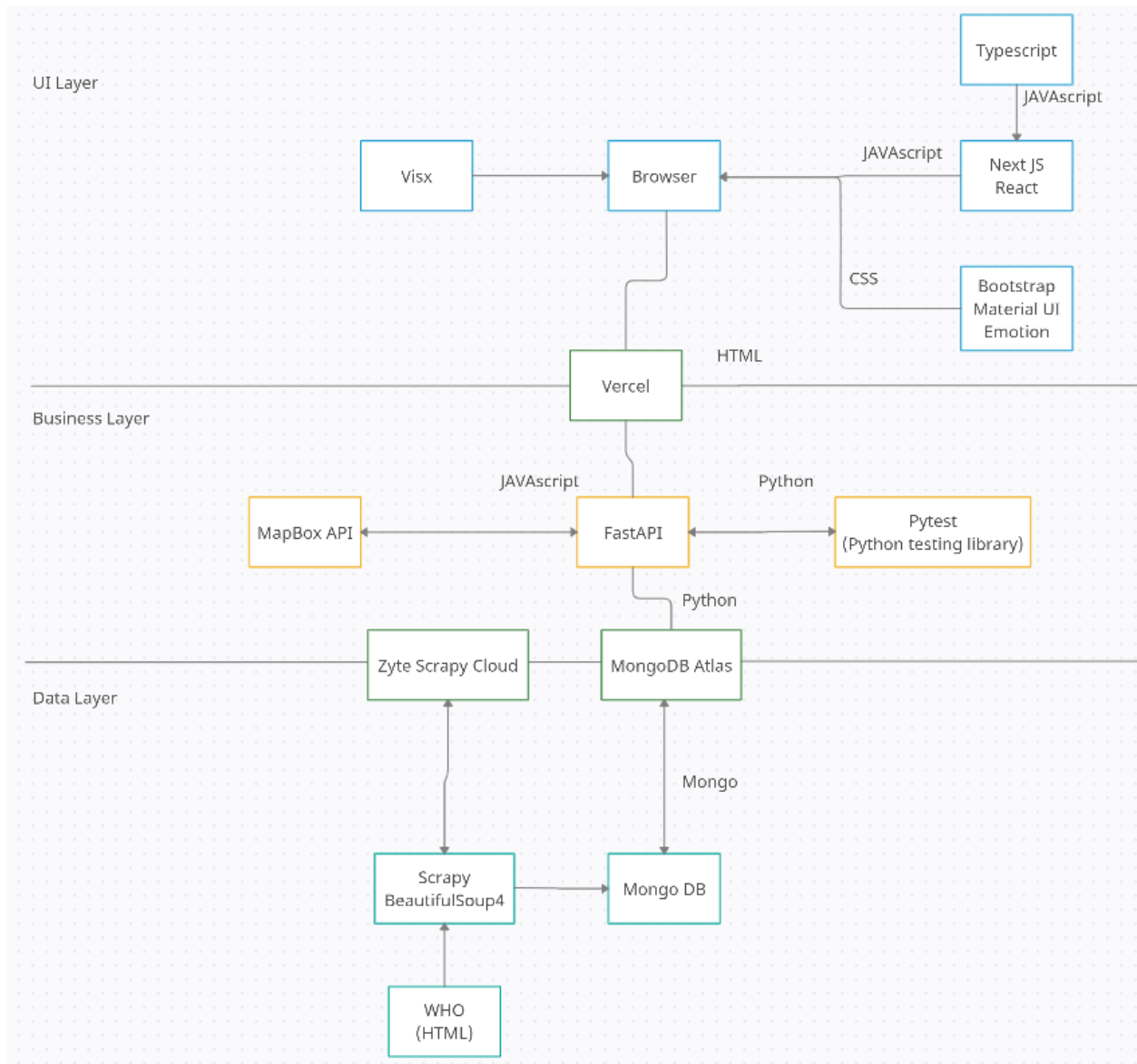
Database

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://app.creately.com/diagram/Cb86r1GgEh8/edit> (WIP) Needs checking + clarification

Project Management

This project is of high importance to us so we have put a great deal of care into making sure we are able to **effectively work together** as a team throughout the term, produce a **high-quality system**, and **stay on top of deliverables**.

Roles & Responsibilities

We have allocated responsibilities to each member to ensure the work is divided equally, optimise our team's efficiency, and allow everyone to sharpen different skill sets.

Role	Member	Responsibilities
Project Manager	Michael Chen	<ul style="list-style-type: none">• Michael is in charge of making sure the project is on track. To do this, he will monitor how each member is going with their tasks and resolve any blockers they encounter to enable them to do their best work.• He will analyse each individual's contribution and provide feedback when necessary to keep the project running smoothly.• On behalf of our team, he will be the person in charge of discussing with the mentor to raise issues, seek feedback, and sort out any uncertainties.• Furthermore, he will set the dates of meetings, decide whether additional ad hoc stand-ups are necessary, and constantly review deliverable specifications.• While the team is working on their tasks for the upcoming deliverable, Michael will be looking ahead to ensure we are also prepared for the next one.
Scrum Master	Jamie Domingo	<ul style="list-style-type: none">• While Michael is in charge of the project from a more high-level perspective, Jamie's duties revolve around the current deliverable and managing weekly sprints throughout the term.• Jamie is responsible for all the project management tools used by the team. She will set up and maintain the team's communication channels, file management space, and sprint board.

		<ul style="list-style-type: none"> ● Jamie will run the weekly meetings, taking notes to keep record of the team's progress, and continuously update the team's living documents to ensure they are always accurate. ● In team meetings, she will conduct a sprint review, encourage a discussion of what tasks were completed, and will execute a sprint planning session to ensure all members are aware of their tasks for the upcoming week.
Product Manager	Jingbang Zhou (Ben)	<ul style="list-style-type: none"> ● Ben is responsible for the overall system that the team will produce. ● He is in charge of reviewing deliverable specifications, monitoring how we meet customer requirements, and ensuring we are providing a positive user experience. ● Ben will conduct meetings with the client when necessary to discuss their needs and how our system can provide them with value. ● Thus, Ben will look at our product from a holistic perspective to ensure we maintain a customer-centric focus and will serve as the team's communication channel with the customer. ● From the team's side, Ben is in charge of ensuring our deliverables are of high quality. He will review our reports, fix formatting issues, and conduct testing according to our user acceptance criteria to improve our system's user experience.
Backend Technical Lead	Cooper Bartlett	<ul style="list-style-type: none"> ● Cooper is the technical lead for the backend of our system. He will make decisions on the backend architecture, adjust it according to customer requirements, and he will also be in charge of setting it up. ● He will inform the team about how the backend communicates with other components, and ensure each team member understands how it works, and how to make valuable code contributions according to their tasks. ● He will sort out any bugs that arise within the backend, help the team with technical issues, and produce explanations and diagrams when necessary.
Frontend Technical Lead	William Sieber	<ul style="list-style-type: none"> ● William and Cooper share similar responsibilities, however, William is responsible for the frontend architecture of our system. ● In addition to setting up the architecture, helping the team make code contributions, and fixing frontend bugs, William will work with Ben, our product manager, to ensure the customer has a smooth and positive user experience.

		<ul style="list-style-type: none"> • Thus, in addition to maintaining the frontend code base, William is in charge of our product's design system and will make sure the customer can use the system according to their needs. • Both Cooper and William will maintain the team's code repository.
Developer	Everyone	<ul style="list-style-type: none"> • As software engineers, every team member will be allocated software tasks. • We will make code contributions to our repository, ensure we are writing clean code and following good software practices. • This includes writing and running tests frequently to ensure our system is working as intended. • We all share the goal and responsibility to build a high-quality system that caters to the customer's needs and provides value to our users.

Project Plans

As a team, we have established a plan to stay on track for this first deliverable. After submission, we will evaluate how it went through a **retrospective** and modify it accordingly. This way, we can make sure that we are constantly **reflecting, optimising how we work together, and setting ourselves up for success**.

Every Monday at 8pm, we will hold a **team meeting**. In this meeting, we will do a **sprint review** of the past week using our **Jira sprint board** and take turns to individually discuss what we have each completed. At the end of this meeting, we will **close the sprint and start planning** the next week ahead. We will also use this time to make preparations for our mentor meeting the following Tuesday at 7pm. We have chosen to work using an **agile approach** because quick, speedy iterations will allow us to **deliver value faster, test early and continuously improve our solution**.

We have decided to use the following project management tools:

- **Jira** will be used to keep track of our **weekly sprints**, conduct our **sprint reviews and sprint planning**, and observe our **project roadmap**.
We will have a **Kanban board** with the following columns: To Do, In Progress, Review, Done. Each deliverable will be represented by an **Epic** consisting of multiple '**Issues**' or tasks. Issues will have a deadline and a person assigned to keep track of it. This will allow us to **visualise our progress** throughout the sprint.
We chose to use Jira as it makes **agile project management** easy to **execute and maintain**. Although it can be quite complicated to learn at first, we decided to use it as it provides plenty of features, can **link to our Github** repository, and is encouraged by the course admin. Furthermore, it is used in industry and gaining exposure to it now would be helpful for us in the future.
- Our **Confluence** space will be used as our **file management system**. This is where we will store our **meeting minutes, brainstorming and drafts**. We will also utilise the Confluence **space**

shortcuts to quickly access our team documents and make the most of the mentions and **notifications platform** to keep each other informed.

We decided to use Confluence as not only was it prescribed to us by the course, it has many **templates** to speed up our workflow. We will use templates such as ‘Sprint Planning’, ‘Meeting Minutes’ and ‘Retrospectives’ to keep our space **neat, consistent and organised**. We will allocate each deliverable a folder to keep track of all relevant documents and notes. We found that Confluence was the best tool to use to maintain our **living documents** as it is easy to update and keep track of.

- For **team communication**, we have decided to use the **Discord** application. This is where we will hold offline discussions and run our weekly meetings. We chose Discord because it is very **lightweight, easy to use**, and everyone already has it. Furthermore, it is extremely convenient that the **voice channel is open 24/7**, as there is no need to schedule and start a call. Through the use of **different channels and threads**, Discord allows us to organise our discussions so that information is not lost and messages are easy to locate.
- **Microsoft Teams** is the prescribed tool to **communicate with our mentor**. We will also use this application to send **calendar invitations** for our sprint reviews and conduct our Tuesday mentor meetings. Since Teams is already connected to our UNSW accounts, we found it the best choice for all **scheduling purposes**.
- **Google Drive** will be used for conducting **collaborative work** with each other, such as rough drafts, brainstorming and individual work. We have chosen to use a combination of Google Drive and Confluence as we believe the features they provide will serve us in different ways. Since Google Drive uses a more **traditional approach** to writing, we will utilise it to curate our **deliverable submissions**. This application was chosen as the team is extremely **familiar** with its interface and we favour **not having to formally publish documents**, unlike Confluence. However, Confluence will still be used for file management and organisation.
- A **Github** repository will allow us to manage our code as the project progresses. We will utilise its features to optimise our **version control, automate our testing and review each other’s work**. We chose this application over Gitlab and BitBucket as we all have Github accounts, we have used it for many **previous projects**, and it can easily be **integrated with our Jira board**.

Software Practices

To ensure our product is high-quality and functional, we have adopted the following good software practices to adhere to.

We will use a separate **‘master’ branch** for our **production-ready code**. We will set up **Github Actions** to automate the building, testing and deployment of our code when branches are merged in. This ensures the code in production will not crash and prevents **code regression**. By automating this, we are also able to work more **efficiently** and can **catch bugs** as soon as they arise.

When completing a **Jira ticket**, we will **identify the issue in our branch name as a prefix**, followed by a concise description of what the code in the branch does. For example, if the Jira ticket was “BT-14”, we would name our branch: “BT-14/description-of-what-branch-does”. Similarly, commit messages will also be written in this format. This allows us to keep track of how our issues are progressing and **keep our**

code organised. We can also **see who is assigned** to the code by reviewing the ticket in Jira. When browsing through our code history, it will be easy to read as the branches will all be written in a **consistent format**.

When we have completed the code on our branch and it is ready to be pushed into master, we will first **create a pull request** so that others can review what we have written. This will ensure that all changes made to our system are **visible** to all team members. Having another person to review it will motivate us to write **cleaner, more readable and optimised code**. Sharing changes, suggestions and explanations will also encourage a **culture of learning** and will ultimately help us all to become better programmers. Only when a team member has **approved** the pull request, most likely one of the technical leads, will the code be **allowed to be merged into master**. Again, Github Actions will ensure that no **merge conflicts** exist, the code is **building successfully**, and it is passing our **automated test suite**.

The technique we will use for merging our code will be the **Squash method**. This ensures that when we review the history, it reflects the changes made by the branches **accurately** and we can **easily roll back** to different versions if needed. We have chosen this technique over Rebasing as we want to avoid having to deal with **complicated history errors** and **confusing merge conflicts**. Although the Rebasing method is linear and more aesthetic, we favour how **precise** the Squash method **visualisation** is in the code history.

Other software practices we will adopt to avoid code smells are:

- Rather than writing everything into one long file, we will **isolate components** and abstract helpers and functions away into **importable modules**. This will prevent **code bloat** as we add more functionality throughout the term.
- We will use a **consistent file naming system** to keep our code **organised**.
- When possible, we will create **reusable functions and components** to avoid **duplicate code**.
- We will use **themes and style files** to ensure our design system is **consistent** and make it easier to create a **positive user experience**.
- Variables will be given **meaningful names** and we will follow **code conventions**, such as initialising constants with uppercase variable names.
- We will ensure that when we are making queries to our API, the queries are **not fetching unnecessary information** and are thus **optimised for performance**.

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	<p>Request Type: GET</p> <p>Description: Gets a list of articles from the data source according to the given parameters.</p> <p>Parameters:</p> <ul style="list-style-type: none">• 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 <p>Response:</p> <ul style="list-style-type: none">• 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

	<p>Testing:</p> <ul style="list-style-type: none"> • 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 <p>Errors (Reason - Status code - Message):</p> <ul style="list-style-type: none"> • 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” <p>Sample response body:</p> <pre>{ "articles": [{ "article": { "url": "www.sampleurl.com", "date_of_publication": "2022-01-01T00:00:00", "headline": "Ebola", "main_text": "Sample main text", "reports": [], "articleId": 1 }]}, "max_articles": 1 }</pre>
/article/{articleId}/content	<p>Request Type: GET</p> <p>Description: Gets a string of text indicating content for a given article.</p> <p>Parameters:</p> <ul style="list-style-type: none"> • articleId - Integer representing the id for a article

	<p>Response:</p> <ul style="list-style-type: none"> content - String detailing the HTML content of an article <p>Testing:</p> <ul style="list-style-type: none"> Ensures response is associated to the correct article Ensures error is raised when called on page with invalid articleId <p>Errors:</p> <ul style="list-style-type: none"> Invalid articleId - 404 (Not Found) - “No article found with given id” <hr/> <p>Sample Response:</p> <pre>{ "content": "Example content response" }</pre>
/article/{articleId}/source	<p>Request Type: GET</p> <p>Description: Gets link for the article for a given article</p> <p>Parameters:</p> <ul style="list-style-type: none"> articleId - Integer representing the id for a article <p>Response:</p> <ul style="list-style-type: none"> link - String detailing the HTML content of an article <p>Testing:</p> <ul style="list-style-type: none"> Ensures response is associated to the correct article Ensures error is raised when called on page with invalid articleId <p>Errors:</p> <ul style="list-style-type: none"> Invalid articleId - 404 (Not Found) - “No article found with given id” <hr/> <p>Sample Response:</p> <pre>{ "link": "abc.testing123.com" }</pre>

/article/{articleId}/assessment	<p>Request Type: GET</p> <p>Description: Gets the WHO assessment for a given article</p> <p>Parameters:</p> <ul style="list-style-type: none"> • articleId - Integer representing the id for a article <p>Response:</p> <ul style="list-style-type: none"> • assessment - String detailing the assessment section for the given article <p>Testing:</p> <ul style="list-style-type: none"> • Ensures response is associated to the correct article • Ensures error is raised when called on page with invalid articleId <p>Errors:</p> <ul style="list-style-type: none"> • Invalid articleId - 404 (Not Found) - “No article found with given id” <p>Sample Response:</p> <pre>{ "assessment": "WHO assessment" }</pre>
/article/{articleId}/advice	<p>Request Type: GET</p> <p>Description: Gets link for the article for a given article</p> <p>Parameters:</p> <ul style="list-style-type: none"> • articleId - Integer representing the id for a article <p>Response:</p> <ul style="list-style-type: none"> • link - String detailing the HTML content of an article <p>Testing:</p> <ul style="list-style-type: none"> • Ensures response is associated to the correct article • Ensures error is raised when called on page with invalid articleId <p>Errors:</p> <ul style="list-style-type: none"> • Invalid articleId - 404 (Not Found) - “No article found with given id”

	<p>Sample Response:</p> <pre>{ "advice": "WHO advice" }</pre>
/article/{articleId}/response	<p>Request Type: GET</p> <p>Description: Gets the WHO response for a given article</p> <p>Parameters:</p> <ul style="list-style-type: none"> articleId - Integer representing the id for a article <p>Response:</p> <ul style="list-style-type: none"> response - String detailing the WHO response of an article <p>Testing:</p> <ul style="list-style-type: none"> Ensures response is associated to the correct article Ensures error is raised when called on page with invalid articleId <p>Errors:</p> <ul style="list-style-type: none"> Invalid articleId - 404 (Not Found) - "No article found with given id" <p>Sample Response:</p> <pre>{ "response": "WHO response" }</pre>
/subscriber	<p>Request Type: POST</p> <p>Description: Sets up an email to receive information whenever a new article is added for a list of given countries</p> <p>Parameters:</p> <ul style="list-style-type: none"> email - String representing email to send to location - List of strings representing countries to be notified on whenever a new article comes out

	<p>Response:</p> <ul style="list-style-type: none"> ● response - String indicating whether or not subscription was successful <p>Testing:</p> <ul style="list-style-type: none"> ● 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 <p>Errors:</p> <ul style="list-style-type: none"> ● Illegal email - 400 (Bad request) - Given email is invalid ● Empty location string - 400 (Bad request) - Location parameter is empty <hr/> <p>Sample Query:</p> <pre>{ "email": "testin123@gmail.com", "location": ["Australia", "New Zealand"] }</pre> <p>Sample Response:</p> <pre>{ "response": "Success, email has been subscribed" }</pre>
/subscriber	<p>Request Type: DELETE</p> <p>Description: Sets up an email to receive information whenever a new article is added for a list of given countries</p> <p>Parameters:</p> <ul style="list-style-type: none"> ● email - String representing email to remove from <p>Response:</p> <ul style="list-style-type: none"> ● response - String indicating whether or not the removal of the

	<p>subscription was successful</p> <p>Testing:</p> <ul style="list-style-type: none">• 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 <p>Errors:</p> <ul style="list-style-type: none">• Illegal email - (400) bad request - Given email is invalid• Email not subscribed - (400) bad request - Given email is not subscribed
	<p>Sample Query:</p> <pre>{ "email": "testin123@gmail.com", }</pre> <p>Sample Response:</p> <pre>{ "response": "Success, email has unsubscribed" }</pre>