Team QQ

SENG3011

DESIGN DETAILS

Deliverable 1

1 CONTENTS

2 OVERVIEW

2.1 BACKGROUND

EPIWATCH is an artificial intelligence-driven epidemic detection system which uses openly available data sources to capture early epidemic signals globally, aiming to prevent the wide-spread of diseases and epidemics altogether. Developed by Integrated Systems for Epidemic Response (ISER), a centre located at UNSW, EPIWATCH utilises its data sources to generate automated early warnings for epidemics worldwide. EPIWATCH provides outbreak alerts, watching briefs, modelling and simulation, real-time decision support, risk analysis tools and rapid epidemic reports.

Through using publicly available data sources, EPIWATCH has been proven to identify outbreak signals earlier than traditional hospital-based surveillance. EPIWATCH relies on the concept that long before health authorities become aware of outbreaks, the public will start talking about them and local news agencies reporting on them. These news reports are surveyed by the EPIWATCH system, and when there are enough reports of an outbreak EPIWATCH will generate a warning.

EPIWATCH was only recently developed (from 2016-2020), but research and modelling suggest that EPIWATCH would have had the capability to detect COVID before it spread worldwide if it had been around earlier. It also could have potentially prevented the spread of many other previous epidemics such as West African Ebola and Mumps.

2.2 OUR GOALS

Given the enormous potential of EPIWATCH to prevent future epidemics and save millions of people, our goal throughout this workshop is to create a web application like EPIWATCH (but slightly simpler). Our application should be able to integrate outbreak data from multiple different sources and present this information in a user-friendly way. It should be able to utilise both news articles (from a given list of websites) and social media posts related to disease outbreaks. It should be able to display the data using visual tools such as graphs and charts, and users should be able to search for information relating to a specific disease. We also aim for our web application to have the capability to analyse the data from the integrated sources and make some predictions. As a more difficult goal, we intend on creating a map of reports which allows users to visualise outbreak locations.

We plan on making our platform open to the public, and similarly to EPIWATCH, through our platform we plan on helping to prevent future outbreaks and epidemics.

2.3 PLANNING

To create our web application, we will need to split the work into different stages. For our web application to be able to integrate reports from different data sources, we will need to utilise several APIs to allow our application to communicate with each of the sources. However for this workshop, we will only be creating a single API, and will be using APIs which already exist and ones that other groups have created. Similarly, other groups will have the option to utilise the API that we create. The workshop can be split into 2 major stages: creating our own REST API and developing our web application.

2.3.1 REST API Stage

For the first stage of the project, we will need to develop our own REST API. The source that our team was given to create a REST API for was CIDRAP (Centre for Infectious Disease Research and Policy). The URL for the CIDRAP source is https://www.cidrap.umn.edu/.

CIDRAP is a centre at the University of Minnesota which aims to prevent illness and death of the general public from infectious diseases. The centre, which was founded in 2001 by Dr. Michael Osterholm, focuses on the *"translation of scientific information into real-world, practical applications, policies, and solutions"*. One of the main strategies used by CIDRAP is to bring together experts to analyse available information and then develop public policy recommendations and guidance. CIDRAP then attempts to communicate this information and make it widely available by publishing news articles containing the information. On the CIDRAP site, the CIDRAP news team provides daily news updates and articles on emerging diseases. They also provide overviews on infectious disease topics.

The articles published to CIDRAP contain disease reports. A disease report is any single mention of a case of a disease. Our REST API needs to utilise the CIDRAP news articles, to provide disease reports on demand. The API will need to be able to identify disease reports from the CIDRAP articles and should use JSON to represent the reports. For all disease reports within CIDRAP articles, the article should specify the type of disease, location, and date. In general, CIDRAP reports on diseases worldwide.

There are 2 stages that will be required for developing this REST API. The first will be to develop a web crawler which scrapes the CIDRAP news articles, and the second stage will be to work on the REST API itself (which will utilise information from the web crawler). The objective of the scraper is to automatically access CIDRAP and fetch the news reports from the articles. The objective of the second stage will be to build a REST API that enables a client to find and access all the disease reports related a given search term. We will also need to publish documentation for the API online.

2.3.2 Web Application Stage

To build the web application, we will utilise our REST API as well as APIs for other sources which will have been created by other groups in similar fashion to us. We are also planning on using other third-party APIs such as (potentially) the Google News API within our web application.

During this stage, we will develop our web application which aim to achieve the goals discussed in section 2.2.

2.4 Features

2.4.1 Must Have

- Scraper pulls information from CIDRAP.
- Public API allows articles and reports to be accessed.
- Frontend allows users to view articles and reports.
- Frontend provides visualisations (graphs, etc.) of reports.

2.4.2 Should Have

- API is not dependent on CIDRAP being online.
- API allows results to be filtered by disease, location and key words.

2.4.3 Nice To Have

- Frontend has a map of reports which allows users to visualise outbreak locations.

3 System Design

3.1 High-Level Design

3.1.1 Overview

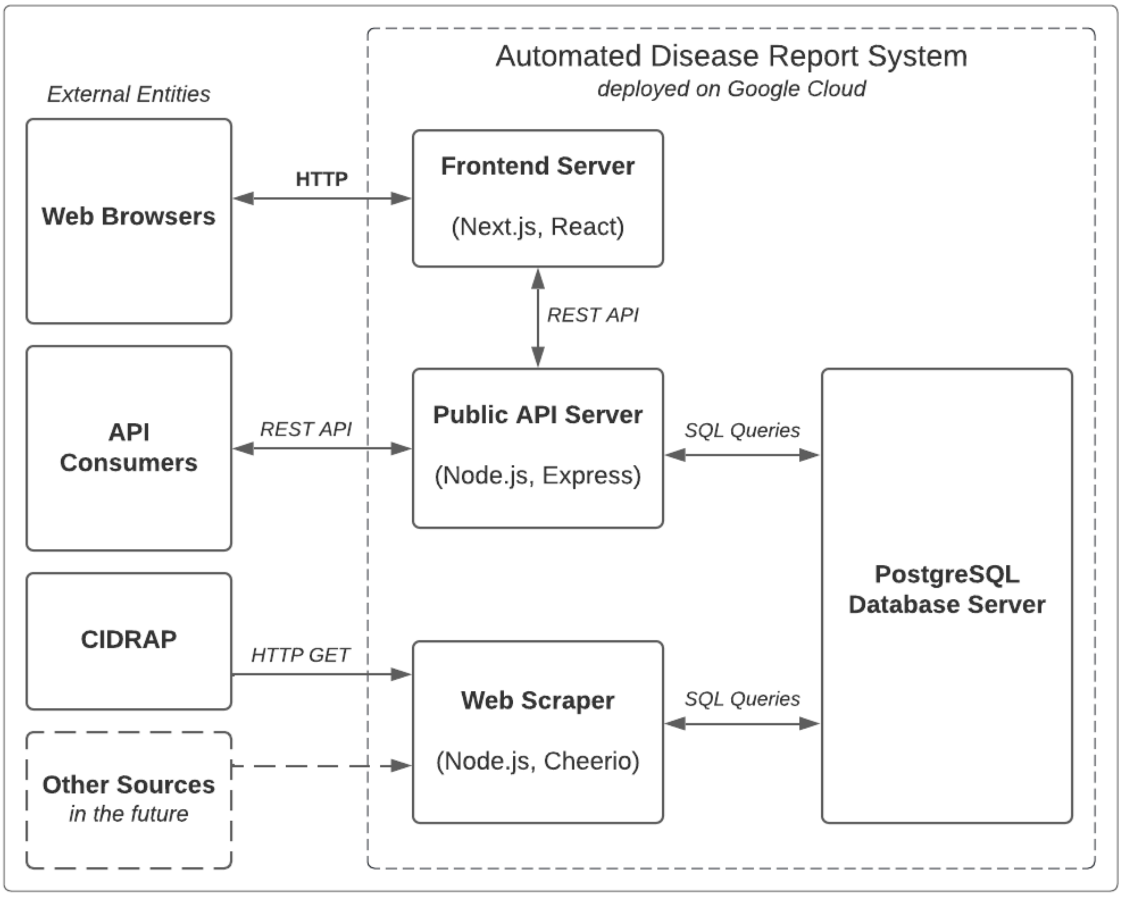
The overall purpose of our system is to extract information from CIDRAP and allow customers to access this information using a Public API we developed. Our entire system relies on the Web Scraper being able to gather information from CIDRAP, where from there it will be processed and available through our Public API and the Frontend Website we create.

Firstly, the Web Scraper will automatically extract data from articles published by CIDRAP using HTTP GET Requests. *Cheerio,* the library we are using for scraping will not produce visual rendering, load CSS or any external resources, or execute JavaScript but more-so parse markup therefore the scraping process will be very quick. The scraped data can then be stored into our PostgreSQL Database automatically as the Scraper will make a connection to the database and automatically execute SQL queries (mainly INSERT) as data is gathered. The PostgreSQL Database is set up so that it will always be online, therefore database calls from both the Web Scraper and Public API Server will be reliable and always return a result.

The Public API Server will also make a connection to the Database and use SQL Queries (mainly SELECT) to extract information out of it. Information gathered by the API Server will be accessed in two different ways; by customers using the API itself and by the Frontend Server. Customers will be able to access information from the Public API Server using REST API methods, namely the GET method. Documentation and instructions on how to use the API will be developed using *Swagger*, where customers will be provided with a well-formatted set of examples and required inputs for each route endpoint. The Frontend Server will use the same available API methods to access information from the Public API Server. Customers will be able to access the Frontend Server using any modern web browser, and information on it will be displayed in a strategic manner to maximise user experience whilst browsing the website.

The Automated Disease Report System components, namely the Web Scraper, Database Server, Public API Server, and Frontend Server will all be deployed using Google Cloud.

3.1.2 Visual System Design



3.1.3 Cohesion

The main method we have used to achieve high cohesion in our design is through the division of each component in the backend and frontend. Cohesion will be increased as each module is only concerned about a smaller subset of the overall problem the system is trying to solve. In the backend, there is a clear distinction between the tasks of persistently storing information and processing the information based on user actions. In our system, these two tasks are separated into the PostgreSQL database and Public API Server. We have also considered the *single-responsibility principle* whilst designing our system to ensure a high level of cohesion is achieved. This principle states that each module in a system should have a single responsibility over a specific part of the system, and it should be entirely responsible for the implementation of part. Adhering to this will ensure that elements in a module are highly cohesive.

3.1.4 Coupling

The only communication between the backend and the frontend is done through a JSON REST API. The JSON API will be designed such that a minimal number of requests will be sent between the frontend and backend to perform tasks. This reduces the coupling between the frontend and the backend as the frontend does not need to know what actions the backend is performing when requests are sent. The backend does not need to know anything about how the frontend is generating the interface or what is causing the requests to be sent, it will just process the JSON requests accordingly.

The modular design of our system will reduce coupling and in turn have many benefits:

- It simplifies the delegating of work for team members as they can be assigned to a specific component of the system.
- The system is more scalable, and in this case one possibility (as visible in 3.1.2) is to add other sources to scrape from.
- Implementation will be made easier, and the codebase is more maintainable and as changes in one component such as the Web Scraper will not have a direct impact on any other components.
- The process of testing and debugging will be simplified as problems can be pinpointed into a specific component rather than the entire system.

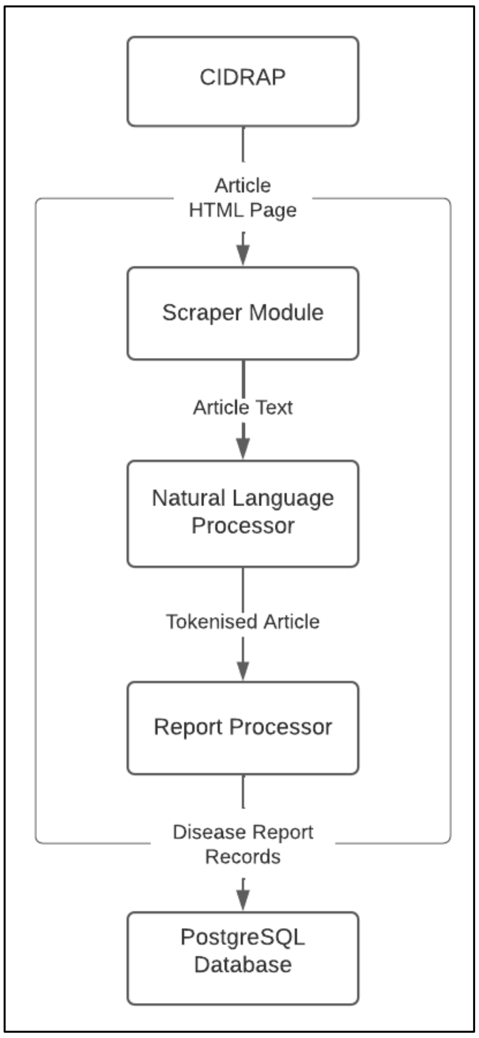
3.2 Web Scraper

The purpose of the web scraper is to scrape and process articles, and to store the disease reports in the database server. The scraper will run every 1-4 hours, keeping the reports up to date without overwhelming CIDRAP or requiring constant processing. The scraper will use HTTP requests to fetch the CIDRAP content, process it, then save the results to the PostgreSQL database server.

This design has many benefits. Having the scraper run autonomously, saving results into the database, means that the system is not dependent on CIDRAP being online. If CIDRAP has scheduled maintenance or is down due to a technical issue, the public API and frontend will still be fully functional. The second benefit to this design is that CIDRAP will only be requested once every few hours. If the API scraped CIDRAP every request, CIDRAP would receive hundreds more requests and it might block our servers for denial of service. This bad design would also cause the scraper to do a lot of redundant work, wasting resources and money. Our system effectively caches articles and reports in the database for immediate access on request.

3.2.1 Pre-release Bulk Scrape

We think it is important for our system to also contain historic disease reports from CIDRAP. To facilitate this, we will write a Node.js script which scrapes a few years' worth of recent CIDRAP articles and processes them. This script will run once offline and will upload its results to the database. This ensures our system is useful from the beginning and it doesn't need to run for months before it is useful.



3.2.2 Scraping Process

The scraper will be split up into three subsystems, the scraping module, the natural language processor and the report processor. The purpose of the scraper module is to traverse the CIDRAP article catalogue and fetch recent article content. It has to extract relevant information from articles such as the title, date, author and main content.

The second subsystem is the natural language processor (NLP). We will likely use an external NLP library because developing a natural language model is a huge task outside the scope of this project. The NLP's task is to take the raw article text and 'tokenise' it. It will output a list of meaningful tokens (words) and can categorize them based on the model.

The final subsystem is the disease report processor. This will take the tokenised article text and compile a list of potential disease reports. These reports will then be forwarded to the database for use in the public API server.

3.2.3 Technologies

3.2.3.1 Node.js

We have chosen to build the scraper using Node.js. Node.js is a backend JavaScript runtime environment which allows rapid development of scalable and distributed systems.

Node.js comes with an extremely useful package manager NPM which provides access to thousands of open-source libraries.

We chose Node.js because it has recently become somewhat of an industry standard for backend server solutions. Although not all members of the team are familiar with it, using it for our project opens up a great opportunity to develop our skills in a tool we might use in the real world. Because of its popularity, Node.js has a rich ecosystem of documentation and guides for almost anything you'd want to do with it. The language's rich library ecosystem means that it only takes a small amount of code and effort to build a complex web system. This is in contrast to a language like C++ which could take teams of developers to make a secure web service from scratch.

3.2.3.2 Cheerio

Cheerio is an open-source library found on NPM which parses HTML markup and allows it to be read and manipulated in a similar manner to jQuery. To extract content, Cheerio simply requires the HTML page as a string. The resulting structure can be queried using jQuery style queries, for example,

```
$('#body').text();
```

returns the content of an element with the id 'body'.

We chose Cheerio because it is a simple to use immediate HTML processing library which is not a web crawler. This is useful because our scraper is built as a cloud function; it is run every hour and it is offline between runs. This design means that our scraper is not a "web crawler;" it does not need to constantly run, updating data in real time. We can simply feed our HTML into cheerio every hour and process the output accordingly.

3.2.3.3 Alternative: Regex for HTML Processing

We chose not to use Regex for the HTML processing because Regex is not built to parse HTML. Regex is best for simple text patterns, while HTML is a very complex markup language. Tools like XML parsers are designed from the ground up to efficiently and intuitively read XML and HTML files and should be preferred. Regex provides too much room for error and makes the code a lot less readable.

3.2.3.4 Alternative: Python with Scrapy

The main alternative technology choice we had was to use Python with the scraping tool Scrapy. Python is an extremely popular, simple and highly readable language. It allows for rapid development, and has a rich ecosystem of documentation and libraries. We decided against Python for a number of reasons. Although we are all well versed in the language, we wanted to challenge ourselves and learn a new, industry-standard language in the form of Next.js.

Additionally, Scrapy is an open-source web crawling framework for Python. Because of its nature as a web crawler, it has to be running on a server all the time, checking for updates to the website. This didn't fit our design as we wanted a scraper that would only run periodically, processing updates and uploading them to the database. These reasons led us to decide against Python and Scrapy.

3.2.3.5 Alternative: Go

Another option was for us to use the language Go for the scraper. Go is an innovative, statically typed and compiled programming language built for web systems, made by Google. Go's main selling point is how it allows users to write highly performant code in a simple language built for multithreading. The main reason we decided against using Go was because none of our team had much experience with it, and it is quite to Python and JavaScript which we are familiar with. Go has a good collection of libraries for scraping but this didn't change our decision to not use the language.

3.3 PostgreSQL Database

We have chosen PostgreSQL to hold the web scraped articles and reports. This is a relational database built for scalability, so it is perfectly suited to holding a large amount of data which will be the product of scraping news articles often.

3.3.1 Database design

The database schema can be found in appendix 6.1. Each article will contain one or more reports, and each report will contain the article URL as a foreign key. The types of each item in the database can be found in appendix 6.2. This design encompasses all data that can be found on the article page and relevant information for each report, including a way to link articles and reports. Often, we are needing to store a list in a database record, but the JavaScript library we are using (Knex) does not have built-in support for arrays. Hence, we have chosen to use a string type but separate each value with commas. This is being used for the syndrome field on the report table and the author field on the article table.

Relational databases come with numerous benefits due to their design, including atomicity and type checking. Through the atomicity of the database, we will be able to ensure there are no concurrency issues between the scraper and the API. Type checking and having constraints on our data will ensure the data is consistent and can be reliably queried. Additionally, a relational database will allow us to perform join operations

3.3.2 Access in Node.js through Knex

To generate and send queries to the database, the scraper and API server will use the Knex library. Knex is a query builder for Node.js which also facilitates connection to most types of SQL database. In Knex, queries are built by calling functions like the "builder" design pattern, as shown below.

```
knex('users')
      .groupBy('count')
      .orderBy('name', 'desc')
      .having('count', '>', 100)
```

This builder pattern is great because it makes the queries more JavaScript friendly. Variables can be easily parsed in as parameters without having to worry about string concatenation or SQL injections. Knex also makes the queries SQL language agnostic. This means the Knex queries work for all SQL languages such as PostgreSQL, MySQL and SQL Server.

Knex also handles connecting to the database server through a secure channel. It creates and manages a "connection pool" when sending requests to the database. This means that instead of starting then dropping connections every time we want to query, Knex maintains a handful of persistent connections which are recycled as needed. This greatly improves the performance and scalability of database operations.

3.3.3 Deployment

The database will be hosted on the Google Cloud Platform (GCP). We will host it on GCP as it simplifies the process of setting up a database. GCP will automatically provision the servers and manage their capacity to ensure we are not wasting any space and can scale up easily. Additionally, Google guarantees 99.95% uptime, so data will always be accessible, increasing the reliability of our API as users can be confident a result will always be returned.

3.3.4 Alternatives

We have chosen a relational database over a NoSQL database for multiple reasons, with the main one being that we want to preserve the relationship between articles and reports. It is important that when a user searches for an article, they are also able to see all the reports that were made within it. A relational SQL database will allow us to perform a join between articles, and reports table, allowing us to easily find reports within articles, or query reports and articles separately.

Some popular NoSQL databases are Firestore by Google, Atlas by MongoDB and DynamoDB by Amazon Web services. Each have their own benefits, Atlas brings with it integrity checks at the level of an SQL database, in that each operation is atomic and you can add type checking and create a schema for the database. DynamoDB is more fluid and a more traditional NoSQL database, with more flexibility and increased scalability. Firestore brings with it easy integration to a Firebase project and by extension the Google Cloud Platform.

A NoSQL database is more suited to storing related data together so that there is no need for joins. If we were to store the reports within each article document, we would lose the ability to query the reports separately from the articles. Alternatively, storing the reports and articles in separate collections but keeping a reference to each other would defeat the purpose of a NoSQL database. Hence, when working with data that has a composition relationship, i.e., a report cannot exist without an article, a relational database supports this most naturally.

SQLite is also an option however has a few downsides when compared with PostgreSQL. SQLite makes queries directly to a file stored on disc, however PostgreSQL makes requests to a server which then reads from a disc. This mean that SQLite lacks the ability to handle simultaneous access by multiple users at any one time, in contrast PostgreSQL was built with multiple user access in mind and has very well-defined permission levels for users to determine which operations are available. PostgreSQL additionally supports more complex queries than other SQL alternatives, in that we could create functions to search the globe by region, or to find countries near to the goal location.

Another SQL alternative is MySQL. Google Cloud Platform has an option for hosting a MySQL however this would come with a couple of downsides to PostgreSQL. PostgreSQL is more suited to applications that will scale up, given we are aiming for our API to be as scalable with a high capacity to handle lots of traffic, MySQL is the weaker choice. We are also a lot more familiar with PostgreSQL because it is taught in the database course at UNSW.

3.4 API SERVER

The API will serve as an interface by which the data that has been scraped can be accessed. It will later serve as a data source for both our and other teams' final applications. It will also be used by the frontend server to fetch disease reports. As such, it is important to consider which technologies will allow us to develop the API both quickly and robustly.

3.4.1 Technologies

A crucial decision in the creation of an API is the choice of the language and web framework with which to construct the service. To make a suitable choice, we considered the positive and negative aspects of various solutions. It was decided that Node.js and Express would be most appropriate for this project.

3.4.1.1 Node.js

JavaScript with Node.js was chosen as the language for the API server. Node provides an event-driven runtime environment that handles asynchronous I/O. This allows multiple clients to concurrently utilise the API and allows multiple services to operate simultaneously. Node.js's V8 engine also provides **just in time (JIT) compilation**, which offers **higher performance** and **lower memory usage** than interpreted languages.

One drawback of Node is that is not well suited to CPU intensive tasks. It was decided that this would not be an issue as there are no significantly intense tasks that will be performed by the API. Another negative aspect is that the asynchronous model of Node can be difficult to understand, however, the scalability and performance of Node outweighs the initial difficulty of programming with asynchronous functions and call backs.

3.4.1.2 Express

Express is an open-source web framework for Node. It is the most **widely used**, industry-standard web framework used with Node, providing high-performance **routing**, HTTP handling (e.g., caching), and **content management**.

3.4.2 Alternatives

3.4.2.1 TypeScript with Node.js

TypeScript is a superset of JavaScript that provides static typing. This augmentation makes it easy to write correct code and avoid runtime errors as types are validated prior to compilation. TypeScript is transpiled into JavaScript, so it is also compatible with Node and its packages.

Due to the nature of working with a type system, development can become harder and require more work. While a type system would be useful for projects with larger scope and more complicated components, the comparatively simple architecture of our system does not require strong typing.

3.4.2.2 Python with Flask

Python and Flask were strongly considered due to their simplicity and our experience with the technologies. Python is an interpreted language with simple syntax. Flask is a lightweight web framework that provides features such as a development/debug server, a Web Server Gateway Interface (WSGI), and a templating engine.

Python allows for a rapid prototyping and development cycle due to its simplicity and comprehensibility. Since Python 3.5, concurrency in python can be handled similarly to JavaScript with async and await functions, as opposed to manual threading. It is slow however due to the overhead caused by its highly abstract nature.

3.4.2.3 Java

Another solution for the creation of an API backend is the use of Java and a web framework such as Spring Boot. Being an object-oriented language, Java allows for the construction of software using object-oriented design. Furthermore, it would be possible to run the API anywhere that has a Java Virtual Machine (JVM).

While Java is far more performant than JavaScript, the simple nature of our project does not call for such a heavyweight tool. Additionally, concurrency between clients would be harder to achieve and would have to be manually threaded, as opposed to the asynchronous function method in JavaScript and Python.

3.4.2.4 C#

Another similar option is to use the C# language with a web framework such as ASP.NET. C# is similar to Java in that it is also designed for use with object-oriented design and that it has static typing. It also has asynchronous function support. ASP.NET is a web framework for C#. One disadvantage of this combination is that to most simply create a .NET application, Windows must be used for development.

3.5 API Documentation Site

Swagger is a tool our group will be utilising to design and document our API. Whilst using Swagger, REST-API documentation is automatically generated in a structured and user-friendly format based on the design, hence any chance of inconsistencies or errors in the documentation are nullified. Previously, creating documentation that is readable takes much of a developer’s time, however the core functionality of Swagger will enable us to avoid this cost overall and focus on the design of our API – the most important part. Since Swagger documentation adheres to a standard format, the API is easily maintainable, and developers can both test and debug problems with ease.

Another benefit of Swagger is that it offers a UI Framework which enables us to interact with the API and gain an insight into how the API reacts to various inputs. Swagger also provides responses in JSON so our API can transfer data using an industry-standard format and be widely accessible to users.

To take advantage of what Swagger has to offer, our group is utilising the design-first methodology where we will use Swagger to design our API before we’ve written any code. Therefore, we can develop a more concise and thought-out API as we will have taken time to think about and prevent any obvious issues regarding the API before facing them during the implementation phase.

3.6 Frontend

The frontend will be built using Next.js and React, with plans to deploy it on the Google Cloud Platform, specifically as a Firebase project. The heart of the frontend server is a Node.js framework called Next.js. Node.js is an open-source JavaScript runtime environment that executes outside the browser. Next.js is a React-based framework that runs within Node.js.

3.6.1 Functionality

The frontend functionality is based around the idea of bringing as much information to the user in the simplest to use and most intuitive UI possible. To do this, we will use different types of visual aids along with various ways to search for more specific information.

3.6.1.1 Visual aids

We are planning to include visual aids such as graphs and maps to help the user visualise the data. Graphs will be used to show the increase of a certain type of disease over a period or any correlation between different diseases. This will help a user quickly see the change in frequency that a disease is being reported. The main type of graph will be a line graph as it will clearly show how reports are being made over the course of time. Additionally, maps will be shown to allow the user to visualise where in the world these reports are being made. As the database will be storing the city as well as the country the report is being made in, we will be able to pinpoint exactly where disease cases are showing up and the user will be getting the most accurate information.

3.6.1.2 Searching

To give as much freedom as possible to explore the information that our API contains, we will introduce various other ways for searching through the data. The basic API routes require time period and location with key_terms as an optional parameter, then we have elected to add in parameters such as source, diseases and syndromes. What we would like to implement with the frontend is the option to make all these parameters optional, allowing a user to find any report based on parameters that do not include time period and location.

In addition to this, we would like to introduce fuzzy searching, where we will match articles and reports based on how closely they match an input string. For example, if searching for a report using the key_term ‘epidemic’, we would also return articles that mentioned ‘pandemic’. Our goal from introducing this feature is to ensure a user captures all relevant articles/reports when they search.

3.6.2 Technologies

3.6.2.1 React

React allows us to write a **declarative**, **component-based** user interface. A declarative UI refers to how we are describing what a component looks like and how its functionality fits into that, rather than how it looks the way it does. With React, state is built into each element and will update the component automatically as state changes. For example, consider the case where the scraper will add more data into the database while the user is looking at the data on the frontend. Using the SWR package, we will automatically fetch new data from the database every few seconds, ensuring that what the user is seeing is most up to date with the database.

3.6.2.2 React Alternatives

There are many alternatives to React, including Angular, Vue, Svelte, or to use no UI library. Angular is a web-application framework by Google which provides a similar feature set to React. We decided against Angular because none of our team has experience with it and it is generally a lot more verbose and bloated compared to react.

Another alternative to React is Vue.js which is yet another JavaScript framework for web development. We decided against Vue because it is a lot less popular than React and is less widely used in the industry. Also, if we were to choose Vue or Angular, we would have to find a Next.js alternative which we did not want to do.

The final alternative to React is to go barebones HTML, CSS and JavaScript. In the modern web development ecosystem, there is little point in building a complex web app from the ground up. By doing so, you are just reinventing the wheel and making development a lot more time consuming and complex. Existing web frameworks significantly speed up development and lead to more readable and maintainable software.

3.6.2.3 Next.js

Another framework we will be using is Next.js, which is a react based framework that aims to solve many problems with basic React. These include **simple ways to compile**, **serve** and **route** pages on a website. Additionally, Next.js provides **static site generation** and **server-side rendering**, which can help to make our site **search engine optimised**. Search engine optimisation refers to the likelihood of our website appearing when people search for related topics, improving how accessible the site is and the traffic on it. Additionally, Next.js is **widely used** and **well documented** which will aid in the development of our site.

3.6.2.4 Next.js Alternatives

One alternative to Next.js would be to use “create-react-app”. “create-react-app” is a command-line tool that automatically configures a development environment for React. It installs and configures the Node.js packages Webpack, Babel and ESLint to kickstart development. “create-react-app” is great for quick prototypes, but for larger projects like our frontend, it restricts and obfuscates important configuration steps. If we used “create-react-app”, we would miss out on a lot of the useful Next.js features like server-side rendering and dynamic routes. Due to these reasons, we decided against “create-react-app”.

Another alternative to Next.js would be to manually set up our own Webpack/Babel environment. Webpack is a module bundler and Babel is a JavaScript compiler. This would restrict us in the same way as “create-react-app” and we would need to install extra packages for server-side rendering and routing. It would also take a lot of effort to set up and configure and would leave more room for error. Next.js handles all of this effortlessly and provides a scalable and maintainable codebase.

Another alternative is the static site generator “Gatsby”. We decided against Gatsby because our site will have a lot of dynamic content which is not supported in a static site.

3.6.2.5 Sass

Sass stands for “Syntactically Awesome Style Sheets” and is a CSS extension language compiled to CSS. Sass provides a lot of useful quality of life improvements such as CSS variables, nested rules and modules. This will make our code more readable and will speed up development time.

3.7 DEVELOPMENT

3.7.1 Source Control

We will use Git as source control for this project. Git is an open-source, source control system that allows multiple developers to work on one codebase simultaneously. We chose Git because it is an industry-standard and we are all very familiar with its workflow. We chose to host the repository on GitHub because it is very widely used and allows teams to contribute to private projects.

3.7.2 Editor

We will use Visual Studio Code as the primary editor for the project. VS Code is an open-source text editor and integrated developer environment (IDE) developed by Microsoft. Although it is not strictly necessary for development, it provides many useful features such as syntax highlighting, error checking and IntelliSense code completion.

3.8 DEPLOYMENT

Our service will be entirely deployed through Google Cloud. Google Cloud is a cloud service provider similar to AWS and provides a full suite of **hosting**, **deployment**, **CI/CD**, **monitoring** and **analytic tools**. Google Cloud offers many services provided either as Software as a Service (SaaS), Platform as a Service (PaaS) or Infrastructure as a Service (IaaS). Once the components have been developed, deployment is as simple as running a "gcloud" command which uploads and deploys the latest version. Google Cloud also handles domains and the generation of SSL certificates, which are needed for encrypted HTTPS traffic.

3.8.1 Google Cloud App Engine

The API server will be deployed through Google Cloud's serverless "App Engine". App Engine is a platform as a service (PaaS) for deploying and hosting web applications in Google-managed data centres. Applications are automatically sandboxed and are run across multiple servers. This means that the server will be automatically elastically scaled when usage increases. It also provides increased redundancy as it would take multiple datacentre failures before all site servers shut down. The platform as a service model is especially helpful for prototyping and rapid development because we don't have to consider purchasing servers or designing a scalable architecture for deployment. We can spend more energy developing the core functionality of the system.

3.8.2 Google Cloud Functions

The scraper will be deployed through Google Cloud serverless functions. Serverless functions allow the deployment of individual tasks which can be triggered by HTTP requests or Google Cloud Schedules. The scraper's functionality is implemented through a single Node.js JavaScript function which is deployed as a Cloud Function. It is triggered automatically by Google Cloud every few hours through Google Cloud Schedules. This means the scraper can run automatically without a server running 24/7, significantly reducing the cost of the system.

3.8.3 Google Cloud SQL

The PostgreSQL database will be hosted using Google's Cloud SQL platform. The platform automatically hosts and manages a distributed PostgreSQL database. This provides good data redundancy and availability because the data is mirrored across multiple servers. Using Cloud SQL means we don't have to worry about setting up our own server to host the database. It also makes the database secure because we do not need to worry about configuring it such that it is inaccessible to the public. The data is also automatically backed up which allows us to roll back to previous snapshots if necessary. The database server comes with 4 vCPUs, 26GB of memory and 100GB of SSD storage, which is plenty for our needs.

3.8.4 Alternative: Amazon Web Services

The main alternative Cloud Provider would be Amazon Web Services (AWS). AWS is arguably more widely used than Google Cloud and provides a similar set of features. We decided against AWS because Google Cloud has a simpler interface which is easier to use for beginners. Since we have little to no experience with Cloud Service Providers, it made sense for us to use a slightly simpler but just as powerful provider.

3.8.5 Alternative: Microsoft Azure

Azure is a cloud computing service operated by Microsoft for application management via Microsoft-managed data centres. Azure is a lot more enterprise-focussed than Google Cloud, which makes it less relevant for our small-scale operation. It also is less open-source than Google Cloud and favours a Windows development environment. For these reasons, it is not the ideal platform for our use case.

3.8.6 Alternative: Virtual Private Server Hosting

Another option would be for us to use a virtual private server (VPS) hosting company like Digital Ocean or Vultr. A VPS is a virtual server hosted by an organisation, usually in the form of a simple Linux server. These servers can be accessed through SSH, and services can be deployed by manually running them on these servers. This approach would be more barebones than using a cloud provider like Google Cloud and has no scalability or automation as is. To make these VPS instances usable, we would have to automate deployment through scripts, containerization or Kubernetes, as analysed below.

3.8.7 Alternative: Cloud Compute Server

Another alternative, similar to VPS hosting would be to use a cloud compute instances such as Google Cloud Compute Engine or AWS EC2 instances. This option has the same pros and cons as VPS hosting, with the added benefit of elasticity. This means that the cloud provider will run your application on a virtual machine with the exact resources it needs. This makes compute instances slightly more scalable and efficient. The cloud provider also provides extra redundancy which reduces downtime. We decided against this option because it has similar problems to VPS servers. It would be too hard to set up and it’s a not very scalable solution.

3.8.8 Alternative: Kubernetes

Kubernetes is an open-source system for automating deployment, scaling and management of containerized applications. Kubernetes clusters coordinate a highly available cluster of computers that work as a single unit. They allow for the deployment of containerized applications to a cluster without tying them to individual machines. Kubernetes would give us more control of the servers our system runs on, allowing us to optimise the best combination of server count, locations and providers to maximise the efficiency of our system. We decided against Kubernetes because no one in our team has experience with it and it would be a lot of work to learn. Google Cloud handles all the hosting and deployment of our applications for us, removing the need for Kubernetes.

4 API Design

4.1 Architecture

Our API will follow a representational state transfer (REST) architecture. REST is an architectural style designed for web resources. It is a client-server architecture that provides a uniform interface for data to be sent and received between a system and a user over the web. RESTful applications have a set of HTTP or HTTPS endpoints that facilitate communication between the client and server. HTTP/HTTPS is the protocol the entire internet is built on, and it defines how to send information over the internet. REST is by far the most common architecture for web APIs and for good reason. Systems that employ a RESTful architecture benefit from increased performance, simplicity, scalability, portability and reliability.

Our API is entirely read-only because its information is gathered entirely using the CIDRAP scraper.

4.1.1 REST Alternatives

Resource-oriented architecture (ROA) is a subcategory of RESTful APIs that model endpoints as a resource hierarchy. Since our data is relatively flat and is read-only to the public, we decided against using an ROA architecture as it provides no benefits to our system. Data is accessed with simple "GET" requests to `/articles` and `/reports`.

GraphQL is another API alternative. GraphQL is completely separate from HTTP and REST, and defines its own system of querying and returning data between the client and the server. From the website, "*GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data.*" In GraphQL, you ask for and receive exactly the data you need. This is good for performance because it reduces the number of queries and stops unnecessary data from being sent from the server. We could not use GraphQL is because our API has to be HTTP and it is outside the scope of the project.

4.2 Endpoint Design

We have adopted a simple API with three endpoints for our system. The endpoints were partially governed by the specification. The input parameters and article/report output format match the specification. We had the freedom to design the endpoints, for which we chose simple, self-explanatory endpoints: `/articles`, `/reports` and `/predictions`. These each return lists of articles, reports and epidemic predictions as described.

4.3 Exceptions

4.3.1 Status Codes

All HTTP endpoints will follow the following standard response status code structure:

Status Code	**Meaning**	**Example**
200	Success	User correctly requests articles
400	Bad Request	User forgets an input query parameter
404	Page not found	User requests an endpoint which doesn't exist
500	Internal Server Error	Our API server has a logic error causing an index out of bounds exception

4.3.2 JSON Response

In the event of an error, the following JSON body will be returned.

```
{ “status”: <status code>, “message”: <error message> }
```

For example, if a user forgets the query parameter `location`, the server will respond with the JSON

```
{ “status”: 400, “message”: “Missing query parameter ‘location’” }
```

4.4 Preliminary API Design

Route Endpoint	**HTTP Method**	**Parameters** *(For GET requests, use query parameters)* *(For POST requests, send as JSON)*	**Result (JSON)**	**Exceptions**	**Description**
`/articles`	`GET`	`{ period_of_interest_start, period_of_interest_end, key_terms, location [optional:] sources }`	`{ articles }` `Where articles is a list of { url, date_of_publication, headline, main_text, reports }`	`400: Missing parameter <parameter>`	Returns a list of articles. `sources` filters by data source, e.g., “CIDRAP“ `period_of_interest` filters by article date. `source` is an optional filter specifying the article’s source e.g. “CIDRAP”
`/reports`	`GET`	`{ period_of_interest_start, period_of_interest_end, key_terms, locations, [optional:] diseases, syndromes source }`	`{ reports }` `Where reports is a list of { diseases, syndromes, event_date, locations }`	`400: Missing parameter <parameter>`	Returns a list of disease reports. `diseases` filters by disease type `syndrome` filters by syndrome type `period_of_interest` filters by report date. `source` is an optional filter specifying the article’s source e.g. “CIDRAP”
`/predictions`	`GET`	`{ [optional:] threshold }`	`{ predictions }` `Where predictions is a list of { disease, syndromes, reports }`	`None`	Returns a list of predicted epidemics based on locations with a sudden increase in specific disease reports. `threshold` is a value from 0 to 1 indicating the minimum probability of an epidemic. 0 means almost no possibility and 1 means high probability.

4.5 Example Http Requests

Example 1: Successful article request.

Route: /articles

HTTP Request

```
HEAD /articles?period_of_interest_start=2019-05-
01T00:00:00&period_of_interest_end=2022-05-
01T00:00:00&key_terms=covid%2Csars&location=china HTTP/1.1
Host: APIDOMAIN.org
Accept-Language: en-us
Accept-Encoding: gzip
Accept:
text/html,text/xml,application/xml,application/xhtml+xml;q=0.9,text/plain;q=0.8,
*/*;q=0.5
User-Agent: Mozilla/5.0
```

HTTP Response

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Referrer-Policy: no-referrer
Content-Length: 626
Date: Thu, 03 Mar 2022 09:12:36 GMT
Connection: Keep-Alive

{“articles”: [
    {
        “url”: “cidrap.umn.edu/news-perspective/2022/03/global-covid-cases-
deaths-drop-except-key-hot-spots”,
        “date_of_publication”, “02 Mar 2022”
        “headline”: “Global COVID cases, deaths drop, except in key hot spots”,
        “main_text”: “COVID-19 surges continue to intensify in locations such as
Hong Kong…”,
        “reports”: [{
            diseases: [“COVID-19”],
            syndromes: [“fever”, “cough”, “tiredness”, “loss of taste”,
                        “loss of smell”],
            event_date: “02 Mar 2022”,
            locations: [“Hong Kong”]
        }]
    }
]}
```

Example 2: Successful reports request.

Route: /reports

HTTP Request

```
HEAD /reports?period_of_interest_start=2019-05-
01T00:00:00&period_of_interest_end=2022-05-01T00:00:00&key_terms=covid%2Csars&
location=hong%20kong HTTP/1.1
Host: APIDOMAIN.org
Accept-Language: en-us
Accept-Encoding: gzip
Accept:
text/html,text/xml,application/xml,application/xhtml+xml;q=0.9,text/plain;q=0.8,
*/*;q=0.5
User-Agent: Mozilla/5.0
```

HTTP Response

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Referrer-Policy: no-referrer
Content-Length: 240
Date: Thu, 03 Mar 2022 09:12:36 GMT
Connection: Keep-Alive

{
    “reports”: [{
        diseases: [“COVID-19”],
        syndromes: [“fever”, “cough”, “tiredness”, “loss of taste”,
                    “loss of smell”],
        event_date: “02 Mar 2022”,
        locations: [“Hong Kong”]
    }]
}
```

Example 3: Bad articles request.

Route: /articles

HTTP Request
```
HEAD /articles?period_of_interest_start=2019-05-
01T00:00:00&period_of_interest_end=2022-05-01T00:00:00 HTTP/1.1
Host: APIDOMAIN.org
Accept-Language: en-us
Accept-Encoding: gzip
Accept:
text/html,text/xml,application/xml,application/xhtml+xml;q=0.9,text/plain;q=0.8,
*/*;q=0.5
User-Agent: Mozilla/5.0
```

HTTP Response
```
HTTP/1.1 400 Bad Request
Content-Type: text/html; charset=UTF-8
Referrer-Policy: no-referrer
Content-Length: 59
Date: Thu, 03 Mar 2022 09:12:36 GMT
Connection: Close

{ “status”: 400, “message”: “Missing parameter ‘location’”}
```

Example 4: Bad reports request.

Route: /reports

HTTP Request
```
HEAD /reports?period_of_interest_start=2019-05-
01T00:00:00&period_of_interest_end=2022-05-01T00:00:00 HTTP/1.1
Host: APIDOMAIN.org
Accept-Language: en-us
Accept-Encoding: gzip
Accept:
text/html,text/xml,application/xml,application/xhtml+xml;q=0.9,text/plain;q=0.8,
*/*;q=0.5
User-Agent: Mozilla/5.0
```

HTTP Response
```
HTTP/1.1 400 Bad Request
Content-Type: text/html; charset=UTF-8
Referrer-Policy: no-referrer
Content-Length: 59
Date: Thu, 03 Mar 2022 09:12:36 GMT
Connection: Close

{ “status”: 400, “message”: “Missing parameter ‘location’”}
```

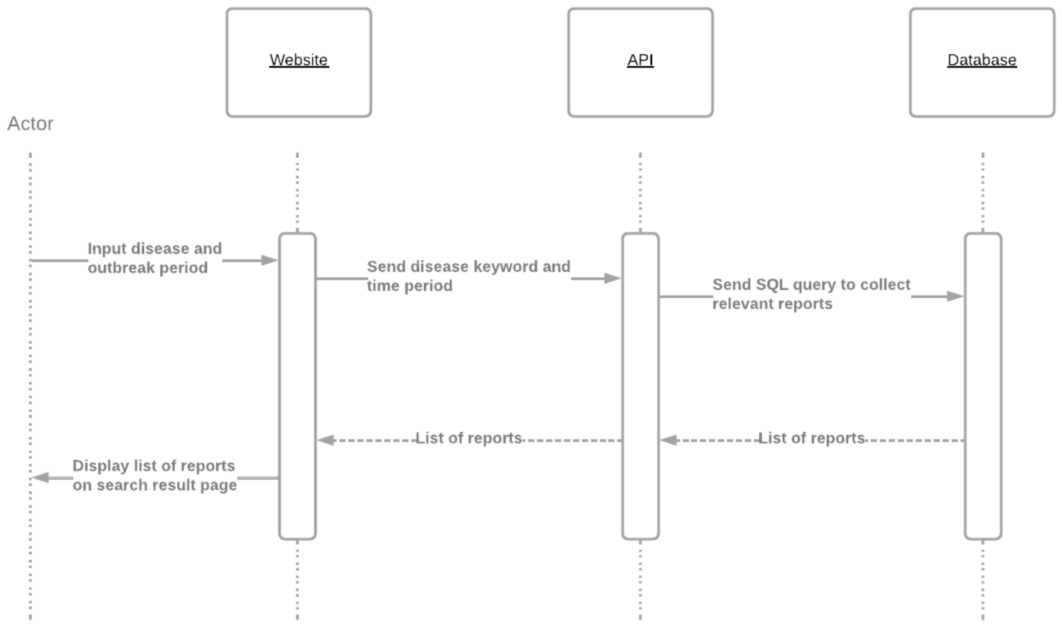
5 User Stories and UML Sequence Diagrams

Feature: User can view reports related to a disease outbreak over a period of time.

As a: UNSW researcher,
I want to: View and filter reports of diseases by period,
So that: I can make predictions about where an outbreak may be occurring.

Scenario: Filter reports by disease and time period.

GIVEN I am on the home page of the site and presented with a list of recent reports,
WHEN I input the name of a disease and a period of time,
THEN I am presented with a list of reports which describe that disease and are within the time period.

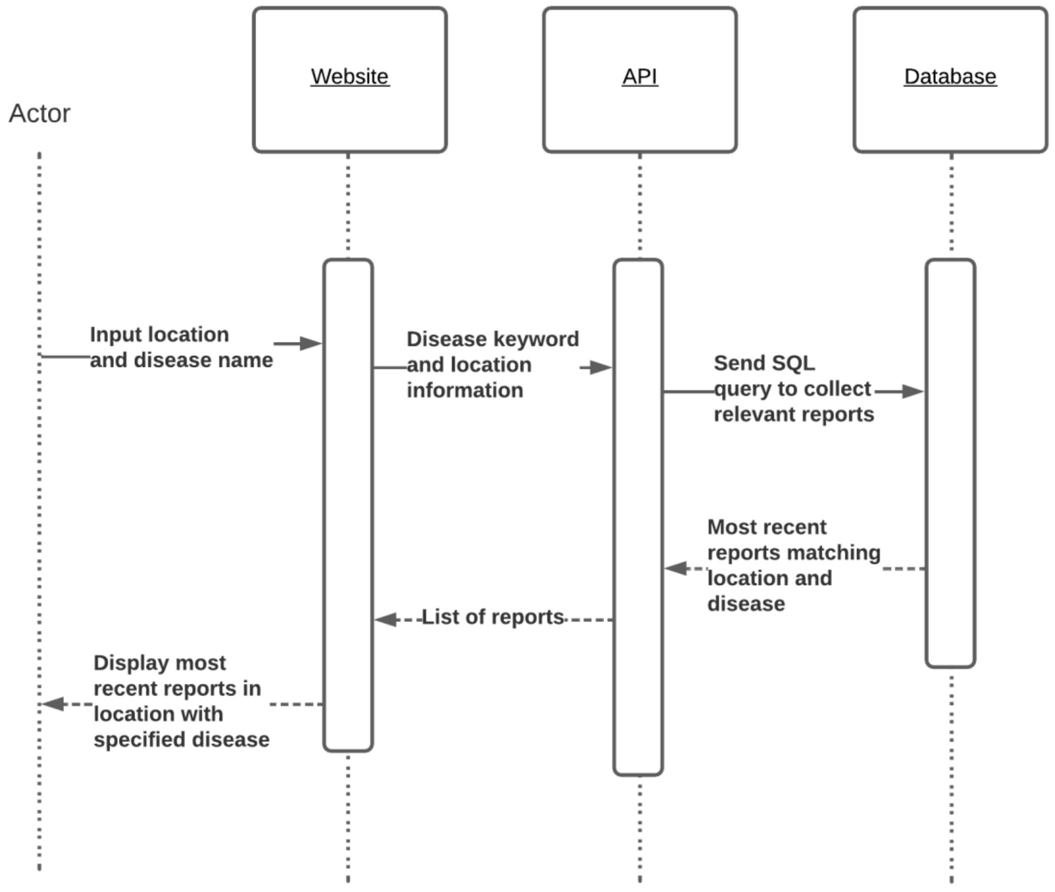

Website
API
Database
Actor
Input disease and outbreak period
Send disease keyword and time period
Send SQL query to collect relevant reports
List of reports
List of reports
Display list of reports on search result page

Feature: User can view reports related to a disease outbreak based on location.

As an: International UNSW student,
I want to: Find news on disease outbreaks in a particular location,
So that: I can easily stay up to date on the health of my home country.

Scenario: Find cases of a disease in a particular country

GIVEN I am on the home page of the site and viewing a list of recent reports
WHEN I enter in my home country name and a specific disease name
THEN I can see all the recent reports of that disease in the country

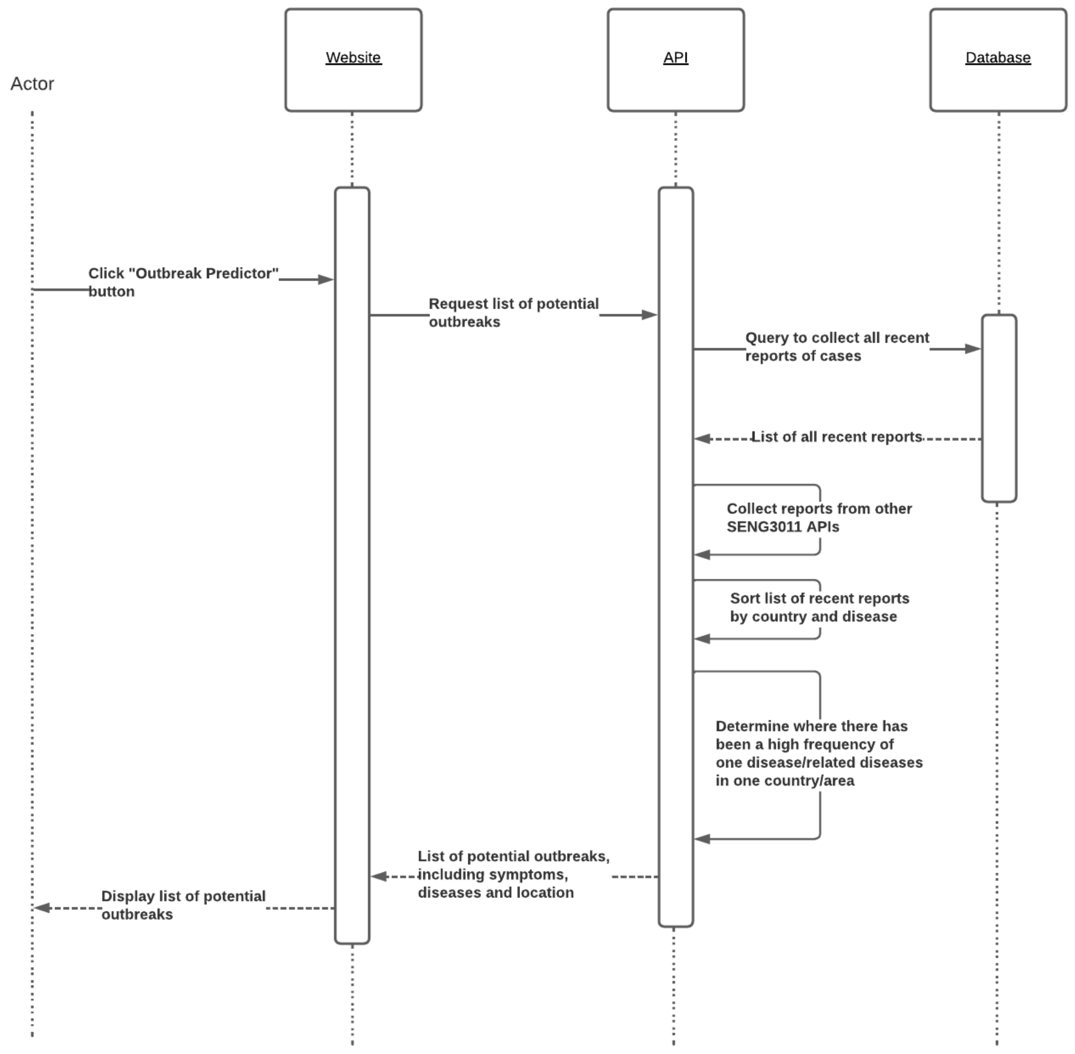

Website
API
Database
Actor
Input location and disease name
Disease keyword and location information
Send SQL query to collect relevant reports
Most recent reports matching location and disease
List of reports
Display most recent reports in location with specified disease

Feature: Predict future outbreaks based on historical data

As a: Doctors without borders employee
I want to: be able to see where possible disease outbreaks are
So that: I can get an idea of where my help will be needed most soon

Scenario: Find a location for where next outbreaks will be

GIVEN I am looking at a list of cases on the site home page
WHEN I click on the "Outbreak predictor" button
THEN I am taken to a page with outbreak predictions

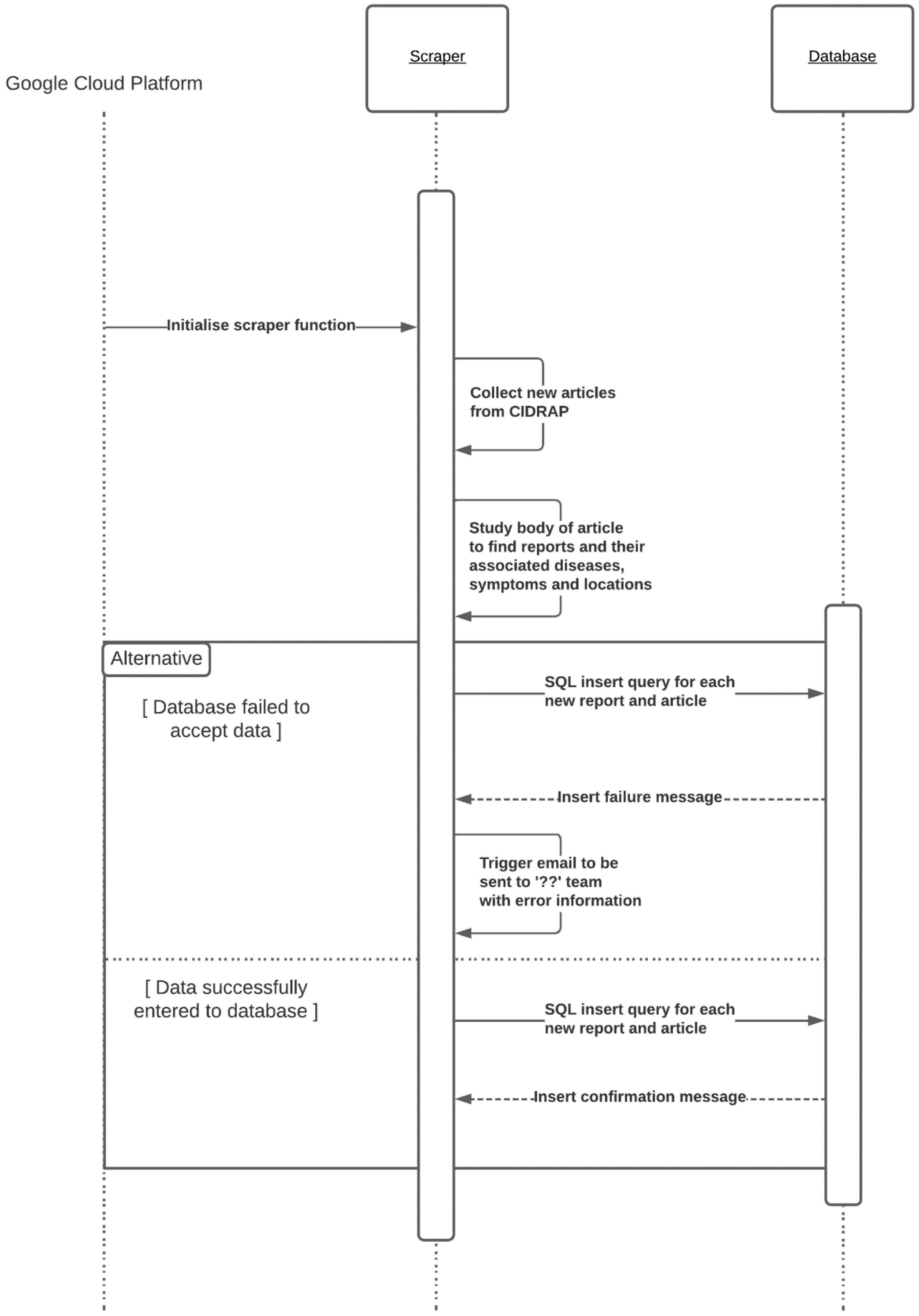

Website
API
Database
Actor
Click "Outbreak Predictor" button
Request list of potential outbreaks
Query to collect all recent reports of cases
List of all recent reports
Collect reports from other SENG3011 APIs
Sort list of recent reports by country and disease
Determine where there has been a high frequency of one disease/related diseases in one country/area
List of potential outbreaks, including symptoms, diseases and location
Display list of potential outbreaks

Feature: Web scraper collects data from CIDRAP each day

As a: UNSW student
I want: the data on this platform to be as up to date as possible
So that: I can stay up to date with disease outbreaks around the world

Scenario: Collect more data from CIDRAP website

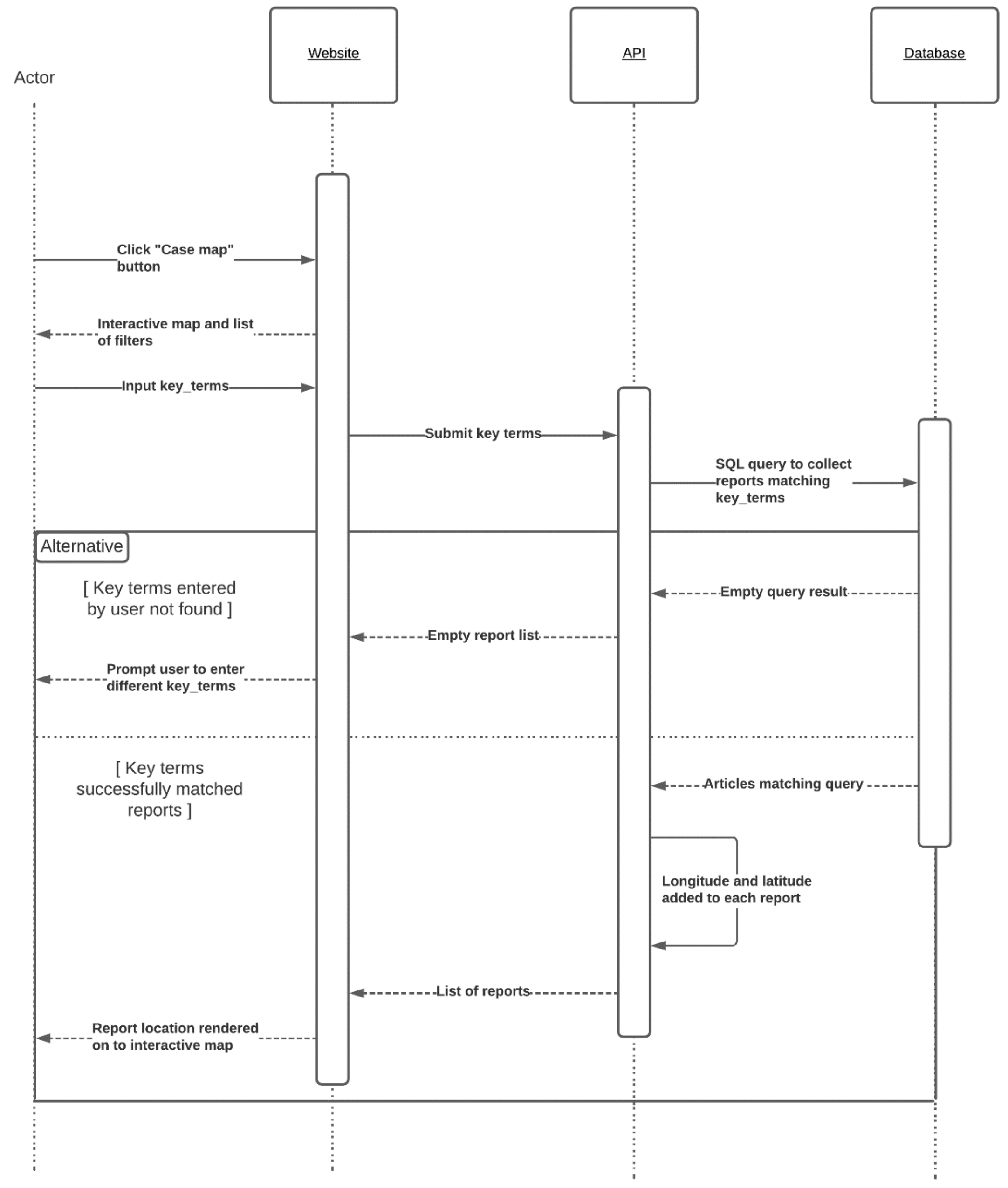
GIVEN the web scraper will be hosted on Google Cloud Platform as a serverless function
WHEN the function is triggered,
THEN the web scraper collects the new articles published from CIDRAP and finds reports within them.


Google Cloud Platform
Scraper
Database
Initialise scraper function
Collect new articles from CIDRAP
Study body of article to find reports and their associated diseases, symptoms and locations
Alternative
[Database failed to accept data]
SQL insert query for each new report and article
Insert failure message
Trigger email to be sent to '??' team with error information
[Data successfully entered to database]
SQL insert query for each new report and article
Insert confirmation message

Feature: Maps to display the location of clusters of a disease

As a: UNSW researcher
I want to: Easily visualise where in the world there are clusters of diseases
So that: I can better understand how communities will be affected by outbreaks

GIVEN I am on the home page of the site
WHEN I click on the button titled "Case map"
THEN I am taken to a page with an interactive map
WHEN I input parameters such as key_terms
THEN I can see relevant reports show up on the map



Website
API
Database
Actor
Click "Case map" button
Interactive map and list of filters
Input key_terms
Submit key terms
SQL query to collect reports matching key_terms
Alternative
[Key terms entered by user not found]
Empty query result
Empty report list
Prompt user to enter different key_terms
[Key terms successfully matched reports]
Articles matching query
Longitude and latitude added to each report
List of reports
Report location rendered on to interactive map

6 TESTING

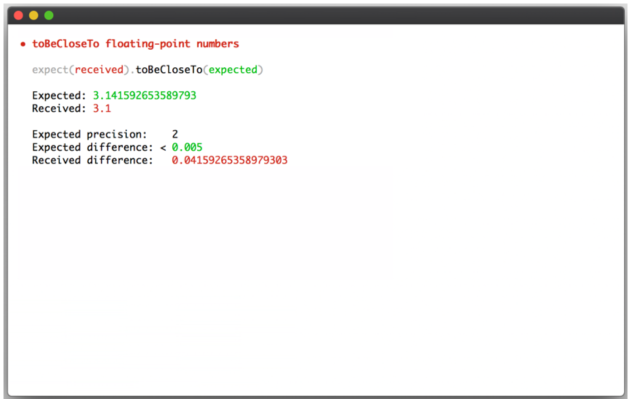
Testing is an important part of building any kind of software. There are various kinds of tests that can be used throughout development, including unit testing, integration testing, and volume testing, each of which verifies software at different levels. Tests are often automated and can be run at times when code should be working, such as prior to building. Another useful aspect of testing is that it makes refactoring code safe, as a failed test can indicate a change in the external behaviour of the program, likely demonstrating a destructive change during the refactor.

One common practice is to implement a new test each time a bug is found. This ensures that the bug can be immediately detected if it were to re-emerge.

A feature of many test suites is the reporting of code coverage. Coverage indicates which lines of code have been run within tests and can provide guidance on which areas might need more thorough tests. While a high coverage percentage is desirable, it is not always practical or necessary to test all lines of code. As such, test writing can be just as programmatically expressive as the working code.

6.1 FRAMEWORK

Most languages have frameworks which automates the running of tests. The testing package we have tentatively chose for this project is 'Jest' which is compatible with many JavaScript environments including Node.js and even frontend frameworks such as React. Jest provides a useful set of expectation functions, including `toBe`, `toBeCloseTo`, `toEqual`, `toStrictEqual`, `toHaveProperty`, `toMatchSnapshot`, `toThrowError`. If an expectation is not satisfied, the framework provides useful feedback upon what was expected and what the true result.



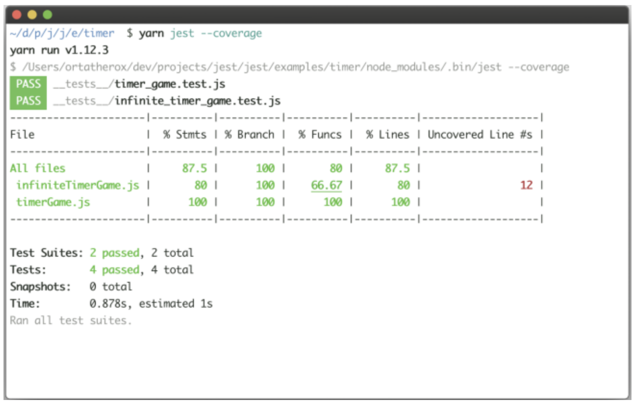
```
• toBeCloseTo floating-point numbers

  expect(received).toBeCloseTo(expected)

  Expected: 3.141592653589793
  Received: 3.1

  Expected precision:    2
  Expected difference: < 0.005
  Received difference:   0.04159265358979303
```

Jest also provides useful coverage information in a user-friendly format.



```
~/d/p/j/j/e/timer  $ yarn jest --coverage
yarn run v1.12.3
$ /Users/ortatherox/dev/projects/jest/jest/examples/timer/node_modules/.bin/jest --coverage
 PASS  __tests__/timer_game.test.js
 PASS  __tests__/infinite_timer_game.test.js
---------------------|----------|----------|----------|----------|-------------------|
File                 |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
---------------------|----------|----------|----------|----------|-------------------|
All files            |     87.5 |      100 |       80 |     87.5 |                   |
 infiniteTimerGame.js |      80 |      100 |    66.67 |       80 |                12 |
 timerGame.js        |      100 |      100 |      100 |      100 |                   |
---------------------|----------|----------|----------|----------|-------------------|

Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.878s, estimated 1s
Ran all test suites.
```

6.2 UNIT TESTING

Unit testing is the most fundamental level of software verification, ensuring that each individual component behaves as expected. The simplest example of a unit test would be an assertion. Considering a simple function, `plusOne(n)`, such a test would follow,

```
assert(plusOne(5), 6);
```

In reality, such unit tests are not performed at runtime as an assert and would instead be run upon build or as part of a workflow.

6.2.1 Method

An example of what a unit test might look for the `plusOne(n)` function like in Jest is,

```
test("Test plusOne function with 5", () => {
  expect(plusOne(5)).toBe(6);
});
```

Properties can be tested to be identical, equal, similar, strictly equal etc. to their expected values or to throw an error/exception.

6.2.2 Expectation examples

Sample 1: Scrape current CIDRAP article list

Function

```
scrapeArticleList(processFn, index)
```

Input

```
processFn: give a callback which saves and checks article info
index: 0
```

Success Condition

processFn is called by `scrapeArticleList once for each article and is parsed the following (sample) article object:`

```
{
    "url": "cidrap.umn.edu/news-perspective/2022/03/global-covid-cases-deaths-
drop-except-key-hot-spots",
    "headline", "Global COVID cases, deaths drop, except in key hot spots",
    "main_text": "COVID-19 surges continue to intensify in locations such as
Hong Kong, where Omicron variant activity got a later foothold",
    "date_of_publication": "02 Mar 2022",
}
```

Sample 2: Process article with COVID report.

Function
```
processArticle(article) -> reports
```
Input
Article Object:
```
{
    “url”: “cidrap.umn.edu/news-perspective/2022/03/global-covid-cases-deaths-
drop-except-key-hot-spots”,
    “headline”, “Global COVID cases, deaths drop, except in key hot spots”,
    “main_text”: “COVID-19 surges continue to intensify in locations such as
Hong Kong, where Omicron variant activity got a later foothold”,
    “date_of_publication”: “02 Mar 2022”,
}
```
Success Condition
Returns a list with one report.
```
[{
    diseases: [“COVID-19”],
    event_date: “02 Mar 2022”,
    locations: [“Hong Kong”]
}]
```

Sample 3: Process article with no reports.

Function
```
processArticle(article) -> reports
```
Input
Article Object:
```
{
    “url”: “cidrap.umn.edu/news-perspective/2022/03/stewardship-us-nursing-
homes-tied-less-antibiotic-use”,
    “headline”, “Stewardship in US nursing homes tied to less antibiotic use”,
    “main_text”: “A new study of more than 400 US nursing homes found that
participating in a quality improvement program that frames antibiotic use as a
patient safety issue was associated with a reduction in antibiotic use and urine
culture collection”,
    “date_of_publication”: “01 Mar 2022”,
}
```
Success Condition
Returns an empty list of reports
```
[]
```

6.2.3 Limitations

While unit tests are effective at ensuring that individual components work separately, they are not appropriate for the verification of a system as a whole.

6.3 Integration Testing

As a system becomes more complex, it is important to make sure that software components are being combined properly to achieve the expected output. Even if all unit tests pass, that does not guarantee that the components have been correctly incorporated.

6.3.1 Method

There are various types of methods for integration testing, including the approaches of, ‘big-bang’, ‘top-down’, ‘bottom-up’, and ‘sandwich’.

The ‘big-bang’ approach is the crudest but simplest and fastest form of integration testing. It involves simply pulling all components together into a complete system. This method is not ideal for complex systems as it can cause it to be difficult to track down the source of an issue.

The ‘top-down’ method involves testing high-level components first and working down the abstraction tree to low level components. This allows for fast prototyping and makes it easy to find the source of errors. The ‘bottom-up’ approach is the reverse of this, where low level components are tested before higher level components. This allows tests to be performed as they are written, unlike the ‘big-bang’ method.

The ‘sandwich’ method is a hybrid between the ‘top-down’ and ‘bottom-up’ methods of integration testing. It involves a target layer, and the levels above and below the target layer. This allows for a thorough test of a particular layer of the software. This method is best suited for large systems and provides high levels of coverage. It is, however, difficult to set up.

The most appropriate method of integration testing for our project, given the short amount of time given for development, is to use the big-bang approach. In the case of our system, the simplicity of the architecture negates the difficulty of finding the source of a bug.

6.3.2 Sample Tests

An example of a test for a successful call to the reports API endpoint in Jest is,

```
describe(“testing-server-routes”, () => {
  it("Successful call to /reports", async () => {
    const { body } = await request(app).get("/reports");
    expect(body).toEqual({
      “reports”: [{
             diseases: [“COVID-19”],
             syndromes: [“fever”, “cough”, “tiredness”,
                     “loss of taste”, “loss of smell”],
             event_date: “02 Mar 2022”,
             locations: [“Hong Kong”]
         }]
      }
    );
  });
});
```

6.4 VOLUME TESTING

Volume testing verifies a range of properties when the system is under high load. When a server is under a high volume of requests, it is important to ensure the maintenance of properties such as,

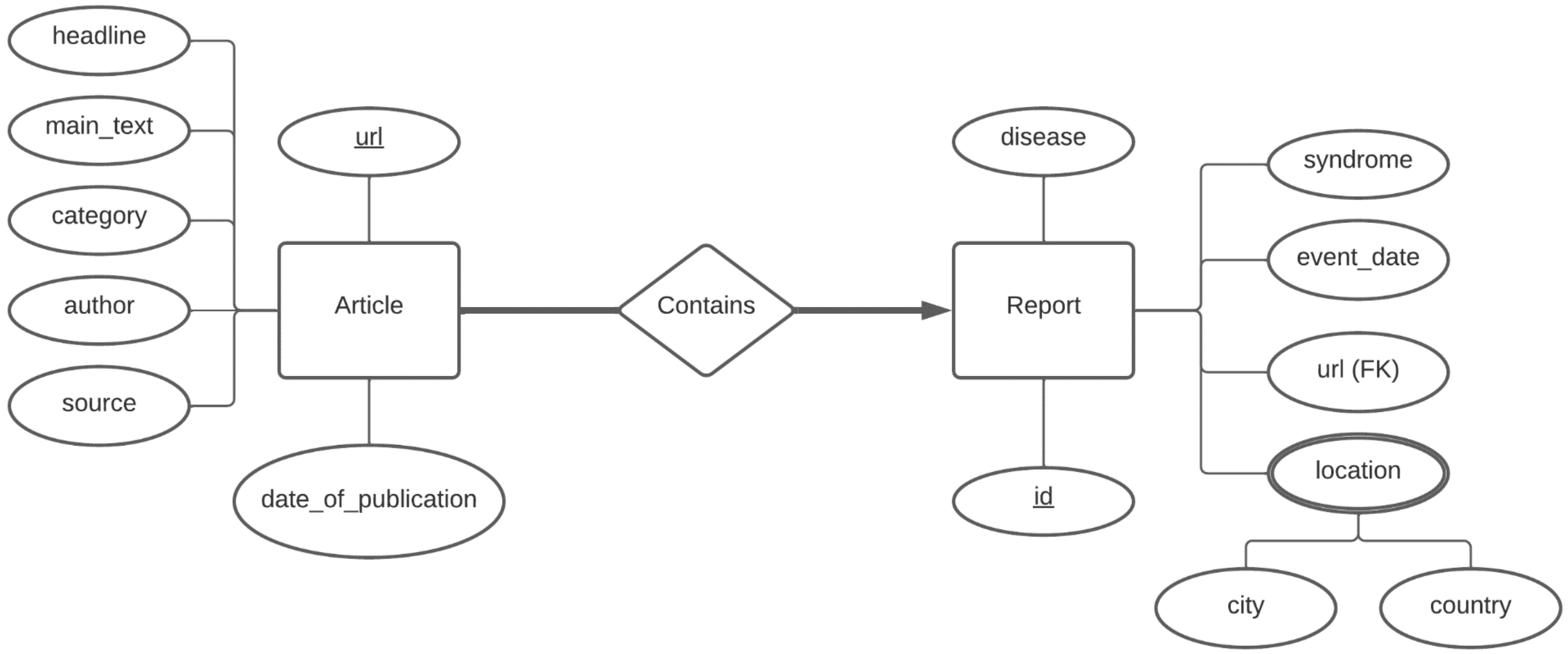
- Data consistency
- Response time
- Error handling
- Resource management

6.4.1 Method

Volume tests are significantly harder to set up than unit tests and integration tests, largely due to the fact that a high volume of requests must be simulated to accurately imitate a real influx of requests.

7 APPENDIX

7.1 DATABASE ENTITY RELATIONSHIP DIAGRAM



7.2 Database Data Types

Table	Attribute	Type	Description
Article	url	unique string	URL of article, primary key of each record. Cannot be NULL.
	headline	string	Headline of article. Cannot be NULL.
	main_text	string	Main body of article. Cannot be NULL.
	category	string	Category of the article.
	source	String	Describes the source of the article, i.e. CIDRAP. Cannot be NULL.
	author	string	Author of the article. If there are multiple authors then they will be in a comma separated list. Cannot be NULL.
	date_of_publication	date	Date the article was published. Cannot be NULL.
Report	id	unique string	Unique random UUID assigned to each report. Cannot be NULL.
	event_date	date	Date the disease was reported. Cannot be NULL. If an article does not specify a date for a report, we will use the date the article was published.
	syndrome	string	List of comma separated symptoms. Cannot be NULL.
	disease	string	The disease the report is relevant to. Cannot be NULL.
	url	string	This is a foreign key, which describes the article this report belongs to. This cannot be NULL.
	city	string	Describes the city the report is in. This is combined with 'country' to produce the multivalued attribute 'location'.
	country	string	Describes the country the report is in. Cannot be NULL.