SENG3011 20T1

Design Details

Medics

Abanob Tawfik   z5075490
Kevin Luxa   z5074984
Lucas Pok   z5122535
Rason Chia   z5084566

Background

EpiWatch, developed by NHMRC's Integrated Systems for Epidemic Response (ISER), is an existing system that monitors and analyses outbreaks. This project will deliver a new system that automates the extraction of outbreak data, which is currently performed manually by the EpiWatch team.

Our system will gather data on the latest outbreaks from the US health department's Centers for Disease Control and Prevention (CDC) website, into a central database. This data will be made valuable to users through a provided web API with extensive search functionality as shown in Figure 1.
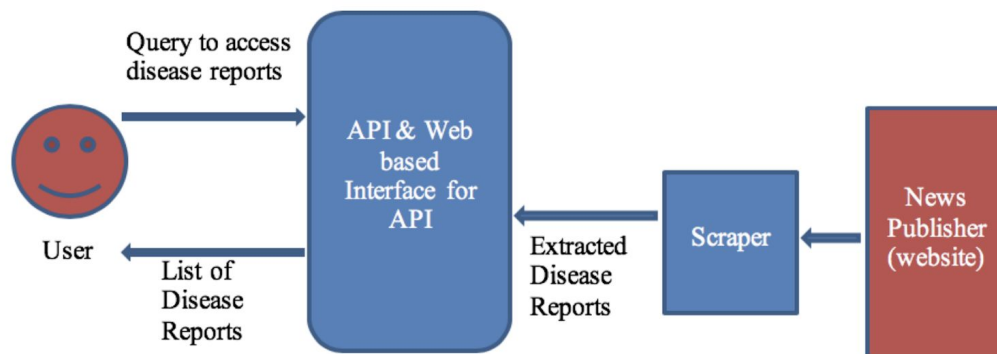

Figure 1: Desired Interaction between user and API endpoint.

1 Initial Documentation

API

Development

We plan to develop the API module by creating a controller that allows access to the reports endpoint. Upon receiving a request, a new instance of the controller is created (handled by ASP.NET), and the database client and services required by the module will be injected into the controller via dependency injection (see the API Source Code Breakdown section for more details). Query parameters will be extracted from the request and used to filter through the database and retrieve articles that match the query. The resulting list of articles is then returned to the user in an OK response. If any of the query parameters are invalid or any of the dates are missing, we will return a Bad Request response alongside an error message to inform the user of the issue.

To aid in debugging our API, we will log each request sent to the API and the outcome of the request. This will help us replicate requests that cause a server error.

Deployment

Our API will be hosted on Microsoft Azure, and will be made accessible through the custom URL https://seng3011medics.azurewebsites.net/api. This will enable any client on the web to use our service to access outbreak reports. Currently, only one endpoint, /reports, is needed to achieve the main functionality of accessing outbreak reports, but we may add more endpoints in the future to provide additional functionality.

While the API service is running, it will be connected to our database of articles, hosted on MongoDB, where all the outbreak articles found by our web scraper will be stored. This will enable our API service to quickly respond to requests, as upon receiving a request from a client, the service can simply query the database for reports that match the given search parameters and return these reports (in JSON form) to the client. The alternative, which is to scrape the website on every request for relevant reports, would be far too slow.

Testing

We plan to test the functionality of our API in a number of ways.

Postman

We will be using Postman, a popular tool used in API testing, to test our endpoint locally before we push it to production. Postman provides a clean, simple interface that allows us to send requests and see the response. Figure 2 shows an example of a request and response in Postman.
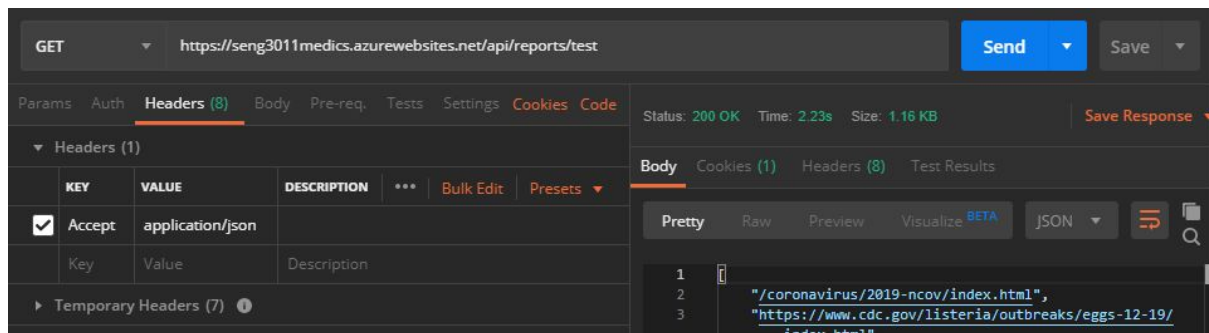
Figure 2: Example of request performed in Postman

Swagger UI

After we have implemented our main API endpoint and pushed it to production, we will test it through our Swagger UI. This will involve passing parameters to our API via our Swagger UI's "try it out" functionality, and comparing the response received with the expected response. We will thoroughly test our API by using different sets of valid inputs (e.g., all parameters provided, optional parameters excluded), as well as some invalid inputs (e.g., invalid date format), as shown in the Example Interactions section.

Unit Testing

We also plan to unit test our API. We will create a script that calls our API with different sets of parameters, and compares the responses from our API with the expected responses. To ensure that the expected responses don't change as we update our scraper to handle more articles, we will create a separate database purely for testing purposes and manually create and insert articles that will enable us to cover a variety of test cases (see the Testing Documentation for more details).

Load Testing

Finally, we plan to load test our API using Loader.io, a load and scalability testing service, to measure our API's response time during artificially high load. This will give us an indication of how much load our API can handle and allow us to accurately place rate limiters to prevent overload.

Details

Parameters

To conform to standard REST API practice, the search and pagination parameters will be passed to our API through URL query parameters in a GET request. The alternative, using the header field, is more complex and is generally used for sensitive parameters (e.g. Authorization) or non-human readable data (e.g. image files).

Endpoint

https://seng3011medics.azurewebsites.net/api/reports

Headers

```
Accept: application/json
```

Parameters

Name	Description	Type/Format	Examples
start_date	The starting time of the period of interest. **Mandatory**	string "yyyy-MM-ddTHH:mm:ss"	"2015-10-01T08:45:10"
end_date	The ending time of the period of interest. **Mandatory**	string "yyyy-MM-ddTHH:mm:ss"	"2015-11-01T19:37:12"
timezone	The time zone associated with the given start and end dates	string	"AEST", "GMT", "PDT"
key_terms		string Comma-separated	"Anthrax,Ebola"
location	The name of a location	string	"Sydney"
max	The number of reports that the user wants to receive	integer (default: 25, maximum: 50)	30
offset	The number of the first report that the user wants to receive (starting at 0)	integer (default: 0)	25

Note: Omitting a search parameter removes the respective search constraint.

Example Interactions

Below are some example interactions with the API. Note that for clarity, requests are not URL-encoded. For example, a space is normally encoded as "%20", but here we leave spaces as they are.

1. Searching for all news of ebola from the Democratic Republic of the Congo in 2017:

Description	In this interaction, the start date, end date, key terms, and location parameters are all used.
Request	GET /reports?start_date=2017-01-01T00:00:00&end_date=2017-12-31T23:59:59&key_terms=Ebola&location=Democratic Republic of the Congo
Response	200 OK {

```
     "articles": [
         {
              "url": "https://www.cdc.gov/vhf/ebola/outbreaks/drc/2017-may.html",
              "date_of_publication": "2017-07-28 xx:xx:xx",
              "headline": "Ebola (Ebola Virus Disease)",
              "main_text": "On May 11, 2017, the Ministry of Public Health of the
Democratic Republic of the Congo notified international public health agencies
of a cluster of suspected cases of Ebola Virus Disease (EVD) in the Likati
health zone of the province of Bas Uélé. ...",
              "reports": [
                  {
                       "event_date": "2017-05-11 xx:xx:xx to 2017-07-02 xx:xx:xx",
                       "locations": [
                           {
                                "country": "Democratic Republic of the Congo",
                                "location": "Bas Uele"
                           }
                       ],
                       "diseases": ["ebola haemorrhagic fever"],
                       "syndromes": ["haemorrhagic fever"]
                  }
              ]
         }
     ]
}
```

2. Search for all news of anthrax in 2018:

Description	In this interaction, just the start date, end date, and key terms parameters are used. This will cause all reports within the period of interest that match the key terms to be returned, regardless of location.
Request	GET /reports?start_date=2018-01-01T00:00:00&end_date=2018-12-31T23:59:59&key_terms=Anthrax

Response	200 OK
	
	{
	    "articles": [
	        {
	            "url": "https://www.cdc.gov/anthrax/outbreaks/2018-apr-18.html",
	            "date_of_publication": "2018-04-20 xx:xx:xx",
	            "headline": "Suspected cases of Anthrax in Turkey",
	            "main_text": "On April 18, 2018, the Ministry of Public Health of Turkey notified international public health agencies of a cluster of suspected cases of Anthrax...",
	            "reports": [
	                {
	                    "event_date": "2018-04-18 xx:xx:xx",
	                    "locations": [
	                        {
	                            "country": "Turkey",

```
                                    "location": "Ankara"
                                }
                            ],
                            "diseases": ["anthrax cutaneous"],
                            "syndromes": ["acute fever and rash"]
                        }
                    ]
                },
                {
                    "url": "https://www.cdc.gov/anthrax/outbreaks/2018-aug-08.html",
                    "date_of_publication": "2018-08-08 xx:xx:xx"
                    "headline": "Outbreak of Anthrax in Sudan"
                    "main_text": "On August 5, 2018, the Ministry of Public Health of
Sudan notified international public health agencies of an outbreak of Anthrax
in its capital city of Khartoum...",
                    "reports": [
                        {
                            "event_date": "2018-08-08 xx:xx:xx",
                            "locations": [
                                {
                                    "country": "Sudan",
                                    "location": "Khartoum"
                                }
                            ],
                            "diseases": ["anthrax inhalation"],
                            "syndromes": ["acute respiratory syndrome"]
                        }
                    ]
                }
            ]
        }
```

3. Search for all news of anthrax, ebola, and zika in 2019:

Description	In this interaction, multiple key terms are used. All reports matching at least one of the terms will be returned.
Request	GET /reports?start_date=2019-01-01T00:00:00&end_date=2019-12-31T23:59:59&key_terms=Anthrax,Ebola,Zika
Response	200 OK ```{ "articles": [{ "url": "https://www.cdc.gov/anthrax/outbreaks/2019-jun-05.html", "date_of_publication": "2019-06-05 xx:xx:xx", "headline": "New cases of Anthrax in Nepal", "main_text": "On April 18, 2018, the Ministry of Public Health of Nepal notified international public health agencies of a cluster of suspected cases of Anthrax...", "reports": [```

```
                    {
                        "event_date": "2019-06-01 xx:xx:xx",
                        "locations": [
                            {
                                "country": "Nepal",
                                "location": "Dharan"
                            }
                        ],
                        "diseases": ["anthrax cutaneous"],
                        "syndromes": ["acute fever and rash"]
                    }
                ]
            },
            {
                "url": "https://www.cdc.gov/zika/outbreaks/2018-aug.html",
                "date_of_publication": "2018-08-08 xx:xx:xx",
                "headline": "Outbreak of Ebola and Zika in Uganda",
                "main_text": "On August 5, 2018, the Ministry of Public Health of
Uganda confirmed that there were two new cases of Ebola and three new cases of
Zika in its capital city of Kampala...",
                "reports": [
                    {
                        "event_date": "2019-08-05 xx:xx:xx",
                        "locations": [
                            {
                                "country": "Uganda",
                                "location": "Kampala"
                            }
                        ],
                        "diseases": ["ebola haemorrhagic fever", "zika"],
                        "syndromes": ["haemorrhagic fever", "acute fever and rash"]
                    }
                ]
            }
        ]
}
```

4. Search for all news from the Democratic Republic of Congo in 2017:

Description	In this interaction, just the start date, end date, and location parameters are used. This will cause all reports within the period of interest from the given location to be returned.
Request	GET /reports?start_date=2017-01-01T00:00:00&end_date=2017-12-31T23:59:59&location=Democratic%20Republic%20of%20the%20Congo

Response	200 OK
	
	{
	    "articles": [
	        {
	            "url": "https://www.cdc.gov/vhf/ebola/outbreaks/drc/2017-may.html",

```
            "date_of_publication": "2017-07-28 xx:xx:xx",
            "headline": "Ebola (Ebola Virus Disease)",
            "main_text": "On May 11, 2017, the Ministry of Public Health of the
Democratic Republic of the Congo notified international public health agencies
of a cluster of suspected cases of Ebola Virus Disease (EVD) in the Likati
health zone of the province of Bas Uélé. ...",
            "reports": [
                {
                    "event_date": "2017-05-11 xx:xx:xx to 2017-07-02 xx:xx:xx",
                    "locations": [
                        {
                            "country": "Democratic Republic of the Congo",
                            "location": "Bas Uele"
                        }
                    ],
                    "diseases": ["ebola haemorrhagic fever"],
                    "syndromes": ["haemorrhagic fever"]
                }
            ]
        },
        {
            "url": "https://www.cdc.gov/anthrax/outbreaks/2017-sep.html",
            "date_of_publication": "2017-09-13 xx:xx:xx",
            "headline": "Outbreak of Anthrax in the Congo",
            "main_text": "On September 13, 2017, the Ministry of Public Health
of the Democratic Republic of the Congo confirmed that there were five new cases
of Anthrax in its capital city of Kinshasa...",
            "reports": [
                {
                    "event_date": "2017-09-12 xx:xx:xx",
                    "locations": [
                        {
                            "country": "Democratic Republic of the Congo",
                            "location": "Kinshasa"
                        }
                    ],
                    "diseases": ["anthrax inhalation"],
                    "syndromes": ["acute respiratory syndrome"]
                }
            ]
        }
    ]
}
```

5. Search for all news from 2017 to 2020:

Description	There may be too many results to be returned in a single JSON response. If the user anticipates that this will be the case, they should select a range of results using the *max* and *offset* query parameters. This will allow the user to retrieve all the desired reports via multiple requests. In the below request, the user retrieves the first 50 results.

Request	GET /reports?start_date=2017-01-01T00:00:00&end_date=2020-12-31T23: 59:59&max=50&offset=0

Response	200 OK
	
	`{`
	`    "articles": [`
	`        ...`
	`    ]`
	`}`

6. Search for all news from 2017 to 2020 (more results):

Description	There may be too many results to be returned in a single JSON response. If the user anticipates that this will be the case, they should select a range of results using the *max* and *offset* query parameters. This will allow the user to retrieve all the desired reports via multiple requests. In the below request, the user retrieves the second group of 50 results.

Request	GET /reports?start_date=2017-01-01T00:00:00&end_date=2020-12-31T23: 59:59&max=50&offset=50

Response	200 OK
	
	`{`
	`    "articles": [`
	`        ...`
	`    ]`
	`}`

7. Not providing an end date

Description	The start date and end date parameters are mandatory. If the request does not include these parameters, an error will be returned.

Request	GET /reports?start_date=2017-01-01T00:00:00&key_terms=Coronavirus&loca tion=China

Response	400 Bad Request
	
	`{`
	`    "errors": {`
	`        "end_date": "required parameter missing"`
	`    }`
	`}`

8. Invalid date format

Description	The start date and end date parameters must follow a specific format (see the Parameters table above). If they do not, an error will be returned.
Request	GET /reports?start_date=2017-01-01&end_date=2017-12-31T23:59:59&key_terms=Ebola&location=Egypt
| Response | ```
400 Bad Request

{
    "errors": {
        "start_date": "invalid value '2017-01-01'"
    }
}
``` |

9. End date before start date

Description	The end date parameter must not be before the start date parameter. If it is, an error will be returned.
Request	GET /reports?start_date=2019-01-31T23:59:59&end_date=2019-01-01T00:00:00&key_terms=Ebola&location=Democratic Republic of the Congo
| Response | ```
400 Bad Request

{
    "errors": {
        "end_date": "end_date is before start_date"
    }
}
``` |

Other Implementation Details

API Language

Our language of choice for implementing our API is C#. Specifically, we will be using the Microsoft ASP.NET Core web framework.

There were several reasons for why we chose the ASP.NET Core framework for our API. ASP.NET is an MVC framework, which means that our code will be separated into controllers which handle API requests, models which store data structures, and services that perform logic work and link the logic to the controller. Using an MVC framework allows for scalable and maintainable development, and the code is organized into separate responsibilities, allowing for functional scaling.

Alternatives

There were a number of alternatives to ASP.NET, including Flask and Node.js. We consider the advantages and disadvantages of each of these below.

Flask

Flask is one of the most popular Python web frameworks. Due to its simplicity and minimality, it is very flexible and it is extremely easy to build a simple web prototype with it. However, Flask is not designed to handle asynchronous programming, which is essential for our API service as it will be retrieving outbreak reports from our database. Furthermore, Flask is not an MVC framework, and for the purpose of creating a project which is scalable in terms of functionality and adding future features or modifying code, an MVC framework is ideal.

Node.js

Node.js is extremely popular, has a large community following and has an enormous amount of support. Its multitude of libraries make development fast, and its compartmentalisation of elements gives it excellent scalability. However, Node.js does not offer the same benefits as ASP.NET. For instance, Node.js does not offer a simple way to connect the database to the API, whereas ASP.NET does. Node.js is also more difficult to host in comparison to ASP.NET, which has many more options for hosting. Our group has also created a project using ASP.NET in the past, giving us more experience in ASP.NET than other frameworks. ASP.NET is backed by Microsoft and has many more support options and community development packages in comparison to Node.js. Whilst both choices had positives and negatives, ASP.NET was a better fit to the technology stack.

Web Scraper

Our web scraper will also be implemented in C#, and will follow the data access process described for CDC.

First, the scraper will access the CDC outbreaks page and filter for tags that contain links to outbreak articles. Each link will then be opened.

An article object will be created for each article by filtering the page for information. The date of publication, headline and main text will be able to be extracted by using the fact that most articles on the CDC website are structured in a similar way, while reports will likely be extracted via pattern matching. We will likely need to update our web scraper and add patterns as we discover different ways in which reports present dates, diseases and locations. To deal with the fact that there may be multiple names for the same disease, the scraper will need to convert disease names to a standard form. For example, we will need to associate "coronavirus" with the standard name "COVID-19".

After the scraper has generated an article object for each article, it will insert them into the database. To avoid unnecessary scraping, we will first check if the article has already been scraped. If the URL of the article already exists in the database and the article has not been updated since the last time it was scraped, we can ignore the article. Otherwise, we will scrape the article as normal and update the entry in the database.

We plan to run the scraper periodically to keep our database up to date with the CDC website.

Database Management System

We need a DBMS for storing outbreak reports. The DBMS we chose was MongoDB, a NoSQL database program. MongoDB is suitable for our project as it uses JSON-like documents, and as described in the project specification, articles and reports are stored in JSON-like objects. Storing reports as JSON-like documents in a MongoDB database would make data storage and retrieval very easy, as the report objects generated by our scraper can be stored directly into the database with minimal changes, and the same objects can be returned by our API. If we used a relational DBMS like MySQL, data associated with a single article would need to be split across multiple tables, due to the fact that an article can contain multiple reports, and we would need to use joins to recombine the data when fulfilling requests. None of this is required when using MongoDB. MongoDB also has some nice hosting options, including MongoDB Atlas, which we plan to use to host our database. From previous experience, it is easy to set up and connect to a MongoDB Atlas cluster, and there is enough free storage to last the duration of the project.

Front-End Language

Our front-end will be implemented using AngularJS, which is a JavaScript framework that componentises the front-end. Componentized development allows us to re-use features without duplicating code, use these components in other projects and unit test specific components. Components also make the code more readable and easier to maintain as changes happen to specific components rather than the whole project. The AngularJS framework also comes with routing, services using observables to subscribe to events, and variable binding to allow data to bind to the view allowing us to use dynamic data on the web page. AngularJS components follow the MVC pattern to an extent, where the TypeScript and directives act as a controller, and the HTML acts as the view. While React offers the same benefits as AngularJS , with the addition of performance, the reason we chose AngularJS over React is that AngularJS code is more standardized; routing, componentization, services, forms

and dependency injection are all provided in the Angular framework, whereas React only offers componentization and the user is left with many options to choose from leading to less standardized code.

Deployment

We chose Microsoft Azure services and DevOps for deployment, as it provides the ideal hosting and deployment environment for our API. Azure services allows us to link our GitHub repository to the web server directly, and by setting up a pipeline for releases on DevOps, we have continuous deployment and integration. Azure services also allowed us to create a staging environment which is a pre-release version for testing, this means that we have the option of creating a stable production release at all times, and a staging environment for testing purposes, upon approval the changes go to production. Azure also offers a release history to allow rollbacks to stable releases. The choice for deployment came down to either AWS or Azure. Due to the nature of our stack consisting of majority Microsoft technology, Azure was the better option for our group, since it also has Microsoft backing and greater support for .NET applications.

Libraries/Packages

We plan to use NuGet packages to obtain libraries that give us access to data structures and tools for development. The following are packages that we will likely use:

HtmlAgilityPack

HtmlAgilityPack comes with a built in web-scraper that fetches the HTML code from a URL, and allows us to perform filtering on the HTML code to retrieve data based on tag elements. We plan to use this library as it would speed up the development process - the time saved from writing an HTML scraper and parser is better spent adding features to our project.

NodaTime

NodaTime is a package for date and time parsing. We plan to use NodaTime as it offers much more capabilities in parsing dates than offered by the default DateTime package in the .NET library.

Swashbuckle

Another package that we may use is Swashbuckle, which automatically generates Swagger documentation from ASP.NET API code. We plan to use Swashbuckle as we expect it will speed up the process of writing Swagger documentation. However, depending on the amount of effort required to add Swashbuckle to our project, it may be less time consuming to manually create all of the Swagger documentation in Swagger UI, especially since we currently only need one API endpoint (/reports). Regardless of how we create the documentation, we will need to add examples to clarify how to use the API.

MongoDB.Driver

MongoDB.Driver is the official .NET driver for MongoDB. This package will enable us to connect to our MongoDB cluster and query our database from inside our C# code.

Other

As the project progresses we plan to use other packages for location parsing, and any data structures needed that would take too much time to implement. All packages can be found in the NuGeT folder found in the dependencies folder.

API Source Code Breakdown

Appendix Figure 1 displays the folder and class structure of our API.

Figure 1: The source code file structure for our project

Endpoints

Our endpoint roots are located inside the Controllers folder. Inside the controller we can add more specific endpoints routes as shown in Figure 4.

```
// GET api/Reports/Test
[HttpGet]
// can change routes
[Route("Test")]
0 references | 0 requests | 0 exceptions
public ActionResult TestEndPoint()
{
    var x = _scraperService.ScrapeData("https://www.cdc.gov/outbreaks/");
    return Ok(x);
}
```

Figure 4: An example of customizing the endpoint with Route attribute

Note that the base of the endpoint is just api/[controller name]. We can create custom paths or modify the routing as required by changing the Route property of each endpoint. To add more endpoints we can either create a new controller if it relates to different functionality, for example an endpoint for users, or we can add more specific actions inside the existing controller and reconfigure the routing.
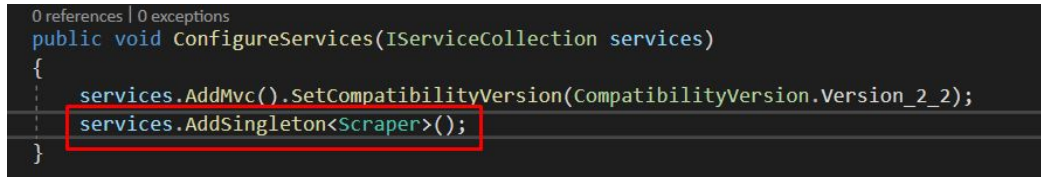
Models

The Models folder contains all of our custom data structures, along with our database context. These will be the objects we return from our endpoints or use in our logic. Our database client will be stored inside the Models folder and will be injected into the controllers as a singleton instance for efficiency.

Services

The Services folder contains code that handles all of the logic. This keeps our controllers free of code and allows a high level view of behaviour for the endpoints. In our case we have the web-scraping logic handled inside one of our services and this can be used in all other controllers.
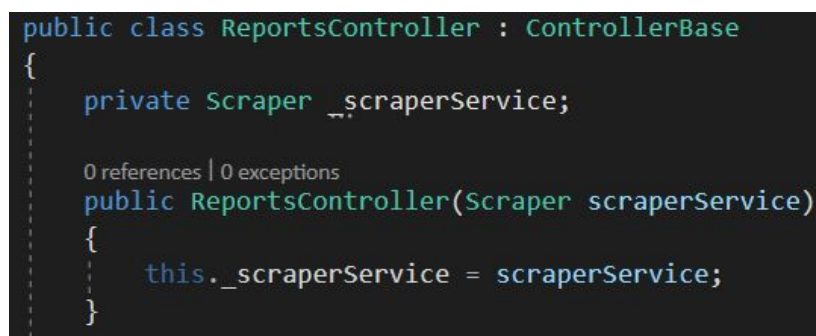
To inject the services into the controllers we will use the dependency injection pattern. This will add either a transient (unique version) or singleton (consistent version) instance of our service across any class that decides to use the service. This is done inside the Startup.cs file shown in Figure 5.

```
0 references | 0 exceptions
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
    services.AddSingleton<Scraper>();
}
```

Figure 5: An example of adding a service as a singleton instance to startup

For controllers to access this service they simply need to add the service as a class field and assign it inside the constructor shown in Figure 6, and the .NET framework will perform the dependency injection on startup.

```
public class ReportsController : ControllerBase
{
    private Scraper _scraperService;

    0 references | 0 exceptions
    public ReportsController(Scraper scraperService)
    {
        this._scraperService = scraperService;
    }
}
```

Figure 6: An example of injecting a service into another class using dependency injection

Configuration

To add settings or configuration variables, such as database connection strings, API rate limit, or authentication tokens, we use our appsettings.json files shown in Figure 7. There are two versions of this settings file - one is for a testing environment and the other is for production.


```json
{
    "Logging": {
        "LogLevel": {
            "Default": "Warning"
        }
    },
    "AllowedHosts": "*",
    "Threshold": 0,
    "ConnectionString": "exampleDBconnectionString;"
}
```

Figure 7: An example appsettings.json file used in the project

We are able to bind these settings to configuration values or even store them inside objects, which means we will not need to hard code static values that need to be used across multiple files. This improves the modularity of our code. An example usage is shown in Figure 8.


```csharp
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
    var connectionString = Configuration.GetSection("ConnectionString").Value;
    services.AddSingleton<Scraper>();
}
```

Figure 8: Binding settings from appsettings.json to variable example

2   Final API Design

Changes from Initial Architecture

During the development process, several changes were made to our initial design to address the challenges involved in implementing the web scraper.

API Endpoint

The API structure has remained mostly unchanged in our final architecture. We still have one controller, ReportsController, which contains one available endpoint, /GetArticles. However, naming changes were made to the route to clarify the behaviour of our API. The base of the endpoint /api/reports is used to access the controller, and the /GetArticles endpoint is now used to retrieve reports contained in articles from the system. We believe that the updated naming convention will make it easier for users to understand the behaviour of the endpoint, and allows greater flexibility in our API design.

Code Structure

The core structure of our code has remained the same and we continue to use the MVC pattern described earlier in the report.

Packages

Throughout the implementation of our API, we discovered that additional packages were needed to perform critical tasks – for example, serialising JSON objects, logging to a file and customising our Swagger UI.

NewtonSoft

NewtonSoft was added for JSON serialization and conversions between JSON strings and objects.

NLog

NLog, a package that adds additional logging functionality, was also added. It offers different logging levels for debugging applications and monitoring the API, enabling us to log to both the console and a file at several information levels (Info, Warning, Error, Trace). NLog was also intuitive to use, simplifying the task of logging.

SwaggerAnnotations

SwaggerAnnotations was required as Swashbuckle did not offer the level of customisation required for our Swagger UI. Although Swashbuckle generated the entire Swagger UI automatically, it lacked the ability to customise parameter descriptions and default values. Using SwaggerAnnotations in conjunction with Swashbuckle granted us much greater control over the automatic generation of the Swagger documentation.

We realised during implementation that a date package was unnecessary as the default library for C# was able to provide the desired functionality. This prompted us to remove the NodaTime package from our project.

Logging

We made sure to log API requests extensively. Every API request will log the request sent in and the parameters received, a successful request will log the items returned from the database, to satisfy the API request. An unsuccessful request will log the details of the errors, and also include potential missing parameters in the request. The time taken for each API call is also logged in order to monitor performance. Appendix A Figure 1 shows the request sent to the API in Postman, and Appendix A Figure 2 shows the log file modification for that request (note that the database was cleared, hence no reports were returned).

Challenges and Shortcomings Addressed

Throughout the process of implementing our API, we came across many challenges and had a number of shortcomings. The majority of challenges were in the implementation, specifically with the web scraper. Many tasks required of the web scraper proved to be more challenging than expected.

Irregularity

The greatest source of challenges while developing our web scraper was irregularity. The CDC website was highly irregular and articles for different diseases often had different formats and layouts. Along with providing no consistent format for their information, the HTML of the CDC articles had no consistent style, making it even harder to search for specific sections of their web pages in our scraper.

Location Extraction

Some outbreaks on the CDC page followed a simple page layout that was consistent and also provided a map indicating the states (US states) in which the outbreak was occurring. Initially, we developed our scraper relying on this specific layout. However, as we began scraping more outbreak pages, we discovered the scraper could not properly extract articles from pages that did not follow this layout. Some pages had no map and provided the locations in the text instead, while pages that did have the map represented the location information in different ways – some pages chose to use a table containing the states, while others listed them out in text.

We overcame this irregularity issue by implementing various fallbacks to enable our scraper to continue extracting locations from articles that did not adhere to the standard format. For example, to parse locations without a map, we developed a regular expression after realising

that location names were always formed by consecutive capitalised words (like "Sydney, New South Wales"). To ignore the false positives such as other proper nouns, we built a LocationMapper service, which determines whether a suspected word is a location using data from several GeoNames CSV files. A surprisingly large number of edge cases arose, such as discerning between a list of locations ("Arizona, Oregon, Rhode Island") and one ("Sydney, New South Wales, Australia"), prioritising US states over countries of the same name ("Georgia"), and handling months of the year being misidentified as cities ("March" being a city in England), to name a few. Extensive unit testing was subsequently done to ensure that the behaviour met desired standards.

Syndrome Extraction

Another major challenge that arose was extracting syndromes from articles. Articles often did not mention any specific syndrome from the syndromes list in the specification, but instead provided a list of symptoms. This required us to determine which syndrome was being referenced based on the symptoms. To resolve this, we created a special mapper that used the syndrome as a key, and the list of symptoms (gathered by us online) associated with those syndromes as the value. Using this mapper we would gather information on both syndromes and symptoms. After scraping the main text and gathering all the symptoms, we would then determine which syndrome is being referenced by selecting the syndrome that had the most matching symptoms in the text. This allowed us to gather information not directly stated in the article.

The irregularity of the articles on the CDC website also made syndrome extraction slightly more challenging, as some outbreak articles listed the signs and symptoms on a separate page, requiring us to collect data across multiple web pages for one article.

Date Extraction

There were many different patterns used to express dates, and thus we needed to create a regular expression and converter for each new pattern that we found. Extracting date ranges was also a challenge, as dates were often scattered around the article. We tried to resolve this issue by searching for the earliest and latest dates in the article, but this method is likely not perfect.

Extraction of Other Information

Processing the main text for the remaining information proved to be a challenge in itself. Often, the article would not use the exact words we were looking for. This issue was resolved through the creation of our special Mapper class. Rather than scanning word by word and checking if the word has special meaning for any of the fields, we checked for the existence of the special keywords in the sentence and added them if we found a match. The mapper also allowed us to handle naming issues, for example "coronavirus" was treated the same as "COVID-19" as per the specification. Our mapper was a simple dictionary containing the name of diseases/symptoms/syndromes as keys, and a list of all possible references as the value. When scanning through sentence by sentence for information, we would first check if the key

is in the sentence, then we would check if any of the references to the key is in the sentence (ignoring duplicates).

By designing a generic mapper and using inheritance, we were able to simply, accurately and efficiently gather specific information from the page relating to syndromes, diseases, symptoms and locations all using one generic method. The use of inheritance allowed us to make tweaks to each specific mapper as required whilst maintaining the default behaviour of the mapper.

Swagger UI

One challenge which took us by surprise was customising the Swagger documentation. Despite Swashbuckle automatically generating Swagger documentation from our controller, little customisation was offered from the Swashbuckle package. In order to add custom annotations, a substantial amount of time was spent on researching the scarce documentary offered in the package. Eventually, we discovered the SwaggerAnnotations package, which gave us more customisation options.

Time Management

There was also considerable difficulty managing the timeline of development. The web scraper was expected to be completed within a week, however due to the issues that arose we spent over double that time. We also required more people to help develop the scraper as the task was not suited to be completed by only a single member of the group. We managed to overcome this issue though communication between group members and the re-assignment of tasks.

Shortcomings

Whilst we overcame many challenges, we had some shortcomings and things we would have done differently.

Firstly, we had planned to run the scraper as a cron job remotely. However, we discovered that our hosting platform offers only a limited amount of free computing time, so we chose to run the scraper locally daily instead in order to save the computing time for the API endpoint.

We had also planned to have more accurate scraping by performing more analysis on CDC articles and seeing which patterns are consistent across multiple outbreak articles, however this was unable to be completed due to a lack of time. If we could have done things differently, we would have spent more time writing a generic scraping method, rather than spending over a week using a more hacky approach, allowing us more time to add extra features.

We had also hoped to include an endpoint for retrieving the number of cases and deaths per for each disease, and an endpoint for travel alerts based on location. However, we ultimately chose to focus our efforts on the scraper instead.

3 Platform Design

Requirements

Priority Key
- **P1 (Priority 1, Must have)**
  - All these requirements form the minimum viable product, crucial to launch
- **P2 (Priority 2, Should have)**
  - All these requirements should be done in the time frame, but are not critical to launch and can be delayed for future release
- **P3 (Priority 3, Could have)**
  - All these requirements are features we would like, but in the time frame we may delay them to future releases to focus on priorities 1 and 2
- **P4 (Priority 4, Would have)**
  - Features we do not expect to have, but show potential extensions of our product.

1. **Querying reports. [P1]**
   1.1. **EpiWatch staff can retrieve reports that satisfy certain criteria. [P1]**
      1.1.1. EpiWatch staff can filter for reports mentioning specific key terms. **[P1]**
         1.1.1.1. Items can be added and removed from the list of terms. **[P1]**
      1.1.2. Staff can filter for reports associated with a particular location. **[P1]**
      1.1.3. Staff can filter for reports in articles published at a certain time. **[P1]**
         1.1.3.1. Available options should include 'past week', 'past month', and a custom date range. **[P1]**
      1.1.4. Staff can filter for reports associated with specific diseases. **[P2]**
         1.1.4.1. Each disease can be included or excluded from the filter. **[P2]**
   1.2. **EpiWatch staff can choose to display the reports on a map. [P1]**
      1.2.1. EpiWatch staff can understand the location and approximate concentration of outbreak occurrences. **[P1]**
         1.2.1.1. The map must show markers at the locations of each report. **[P1]**
         1.2.1.2. The markers should allow staff to distinguish the concentration of reports associated with a general location. **[P2]**
      1.2.2. Staff can view the reports around a particular location. **[P1]**
         1.2.2.1. Upon selecting a marker, a list of reports at the respective location should be displayed. **[P1]**
         1.2.2.2. Details of a specific report can be viewed (see *req. 1.3*). **[P1]**
      1.2.3. Staff can interact with the map. **[P1]**
         1.2.3.1. Ability to zoom in and out, and pan around. **[P1]**
   1.3. **EpiWatch staff can view the details of a specific report. [P1]**
      1.3.1. EpiWatch staff can view the report's associated diseases, and its parent article's headline, publication date, source, and preview of the text. **[P1]**
      1.3.2. Staff can access the page of the original article. **[P1]**
      1.3.3. Staff can access a model of the disease's development based on the reported case numbers of the article (see *requirement 2.1.5*). **[P1]**

1.4. External developers can also retrieve public report data from the system. [P1]

    1.4.1. External developers can retrieve reports satisfying certain criteria. **[P1]**

        1.4.1.1. Such criteria should include article publication date range, key terms mentioned, and associated locations. **[P1]**

        1.4.1.2. Involves sending a request to a standard programming interface.

        1.4.1.3. Invalid requests should be met with meaningful responses. **[P1]**

        1.4.1.4. Pagination parameters should be available. **[P1]**

    1.4.2. External developers can learn about the usage of the interface. **[P1]**

        1.4.2.1. Documentation detailing the request and response formats should be accessible. **[P1]**

2. Modelling epidemics. [P1]

2.1. EpiWatch staff can predict the progression of an epidemic using a deterministic SIR model. [P1]

    2.1.1. EpiWatch staff can view the actual reported case numbers. **[P1]**

        2.1.1.1. The total reported cases on a specific date can be examined. **[P1]**

    2.1.2. Staff can view the predicted case numbers. **[P1]**

        2.1.2.1. Involves the size of each SIR compartment on a certain date. **[P1]**

    2.1.3. Staff can compare the reported cases with the prediction. **[P1]**

        2.1.3.1. Requires a visual representation of the reported data combined with the modelled growth. **[P1]**

        2.1.3.2. Compartments can be included or excluded from the display.

    2.1.4. Staff can adjust the modelling parameters to alter the prediction. **[P1]**

        2.1.4.1. Must include transmission parameters (population size, basic reproduction number R_0) and response parameters (case fatality rate, recovery time, R_0 reduction). **[P1]**

    2.1.5. Staff can view a model auto-generated from reported case data. **[P1]**

        2.1.5.1. Parameters should be automatically adjusted based on reported cumulative case numbers to find a close prediction. **[P1]**

        2.1.5.2. Data including deaths and hospitalised cases should improve the quality of the automatic prediction, if available. **[P2]**

2.2. EpiWatch staff can switch between different models. [P2]

    2.2.1. EpiWatch staff can save their current parameter configuration. **[P2]**

    2.2.2. Staff can load and delete existing parameter configurations to switch between different models. **[P2]**

3. Managing data catalogue. [P3]

3.1. EpiWatch staff can manually update articles stored in the system. [P3]

    3.1.1. Logged-in staff members can add and remove articles. **[P3]**

    3.1.2. Logged-in staff can update the details of an existing article. **[P3]**

        3.1.2.1. Details include headline, publication date, source and text. **[P3]**

    3.1.3. Staff can update the list of reports associated with the article. **[P3]**

        3.1.3.1. See *requirement 3.2*.

    3.1.4. Logged-in staff can view a priority feed of newly-scraped and updated articles, which are probable candidates for manual review. **[P3]**

             3.1.4.1.   Articles can be manually dismissed from the priority list. **[P3]**

    3.2.   **EpiWatch staff can manually update reports stored in the system. [P3]**

        3.2.1.   Logged-in staff members can add and remove reports. **[P3]**

        3.2.2.   Logged-in staff can update the diseases, syndromes and cases associated with a report. **[P3]**

             3.2.2.1.   Ability to add and remove entries. **[P3]**

4.   **All staff members have an account on the system. [P3]**

    4.1.   Requires a database to store account details such as name, username, administrator status and (encrypted) password. **[P3]**

    4.2.   EpiWatch administrators are able to manage accounts on the system. **[P3]**

        4.2.1.   Administrators can create a new account for a staff member. **[P3]**

             4.2.1.1.   Fields must include username and default password. **[P3]**

        4.2.2.   Administrators can update and delete existing user accounts. **[P3]**

    4.3.   EpiWatch staff can log in to their account with their username & password. **[P3]**

        4.3.1.   Requires a login form. **[P3]**

        4.3.2.   Requires a server and database to authenticate users (see *req 4.1*). **[P3]**

    4.4.   Every user has a profile and can update their personal details, such as their name and password. **[P3]**

        4.4.1.   Requires a profile page and options to update and save details. **[P3]**

        4.4.2.   Requires a database to store and update users' details. **[P3]**

Use Cases

The following use cases cover all Priority 1 requirements as well as some Priority 2 requirements.

Phase 1 (Backend / Infrastructure)

1. EPIC: Data Extraction and Storage:
   1.1 REQ Extract outbreak information from pages of varying formats automatically using a Scraper ✅

Use Case 1.1.1	Navigate through CDC RSS feed and return outbreaks data in JSON✅
Users	Internal - Backend C#
Overview	A Reusable Web Scraper that looks for html tags on CDC website to get relevant outbreak information.
Category	Phase 1 Backend - Data Extraction and Storage (1)
Trigger	Cron Job from Azure at Midday and Midnight
Precondition	Executed Cron Job AND (Midday OR Midnight)
Postcondition	JSON extract of html data from CDC Website

Use Case 1.1.2	Fits JSON outbreaks data into article class and returns a list of articles ✅
Users	Internal - Backend C#
Overview	Retrieves JSON data from scraper and inserts this into Article & Object objects. Article Objects must have: - Url of outbreak page - Date of publication - Headline & main text - Report Object (Date of Event, Location, Symptoms)
Category	Phase 1 Backend - Data Extraction and Storage (1)
Trigger	Completion of 1.1.1
Precondition	JSON data returned from web scraper defined in 1.1.1
Postcondition	Array List of Article Objects

1.2 REQ: Store extracted information in a database in a standard form, to be retrieved when needed ✅

Use Case 1.2.1	Cycle through articles and store into mongoDB ✅
Users	Internal - Backend C#
Overview	Cycle through the Array List of Article Objects produced in 1.1.2, insert article object into Mongo DB
Category	Phase 1 Backend - Data Extraction and Storage (1)
Trigger	Completion of 1.1.2
Precondition	Array List of Article Objects is not empty
Postcondition	MongoDB Articles Database is updated with latest outbreaks information

2. EPIC: Data Retrieval via REST API:

2.1 REQ: API must be able to search based on date range, key terms, and location then return this to frontend in JSON ✅

Use Case 2.1.1	URL Pagination of the parameters is received by backend ✅
Users	API Users & Indirectly: Frontend Platform Users
Overview	The parameters for API to search on must be passed through the url of the api using url pagination. Backend must recognise and extract these criteria when searching through DB. An example: .../reports?start_date=2018-01-01T00:00:00&end_date=2018-12-31T23: 59:59&key_terms=Anthrax
Category	Phase 1 Backend - Data Retrieval via REST API (2)
Trigger	Valid API GET Request Received
Precondition	MongoDB Articles Table is not empty
Postcondition	Updated articles information returned in JSON

2.2 REQ: Formal Swagger Documentation that describes the purpose of the endpoint and also instructs users how to use the Swagger Endpoint ✅

Use Case 2.2.1	Clear descriptions and parameters definitions with examples for of input and output fields ✅
Users	API Users
Overview	Swagger Documentation to begin with an introduction summary of the API purpose and how it is used. Each parameter to have a Title and Description Field that describes the purpose and usage of the parameter.
Category	Phase 1 Backend - Data Retrieval via REST API (2)
Trigger	Persistent
Precondition	NIL
Postcondition	MongoDB Articles Database is updated with latest outbreaks information

Use Case 2.2.2	Indication of Required Fields and Try It Out Section with a preloaded example ✅
Users	API Users
Overview	Swagger Documentation must indicate where a parameter is required for the GET Request to work. There must also be a Try It Out Section which has a pre-defined example request which they can click to execute. Upon executing, the api will return JSON result which is shown directly on the swagger documentation.
Category	Phase 1 Backend - Data Retrieval via REST API (2)
Trigger	Persistent
Precondition	NIL
Postcondition	Users know which parameters are required when using the API

Phase 2 (Frontend)

3. **EpiWatch staff can choose to display the reports on a map.**

Use Case 3.1.1	Upon loading, the map will show location pin for each location where is at least 1 outbreak retrieved from cdc and other sources ✔
Users	Frontend Platform Users - General Population and EpiWatch Team
Overview	Location pins will show where there have been outbreaks. When selected, a small modal will popup to show a list of outbreaks for that specific location. Location pins are consolidated on the frontend. I.e. Where there are 5 articles in the same city, then the frontend will only show 1 location pin.
Category	Phase 2 Frontend - Main Page
Trigger	Location Pin being generated
Precondition	MongoDB articles table is not empty
Postcondition	Users are shown specific location pins which displays where outbreaks are located

Use Case 3.1.2	List relevant articles with headings when a location pin is selected ✔
Users	Frontend Platform Users - General Population and EpiWatch Team
Overview	Upon clicking on the location pin, relevant article heading and link is presented to the user. Where there are multiple articles, the modal is presented as a scrollable modal. Multiple articles will appear separate in the scrollable modal.
Category	Phase 2 Frontend - Main Page
Trigger	User clicks on location pin
Precondition	MongoDB articles table is not empty AND location pin is valid
Postcondition	Users are presented with the articles related to the selected pin

Use Case 3.1.3	Detailed Modal will popup when article heading is clicked on. ✅
Users	Frontend Platform Users - General Population and EpiWatch Team
Overview	Upon clicking on the article heading, a large detailed modal will popup. This modal will close when clicking outside the modal for when users wish to close the modal. Modal will display title of article, main text of article, date and its Source as well as a link to original source page.
Category	Phase 2 Frontend - Main Page
Trigger	User clicks on article heading
Precondition	Article is not null and Article is valid
Postcondition	Users are presented with the detail information that is in the article

Use Case 3.1.4	Search options by keywords, location and time period can be inputted on the top left of the map ✅
Users	Frontend Platform Users - General Population and EpiWatch Team
Overview	Upon searching, the map will be updated to show only outbreaks relating to the criteria inputted. A list of relevant articles will be displayed upon clicking on the location pins. This will also filter the list view of articles.
Category	Phase 2 Frontend - SearchPage
Trigger	User clicks on search button
Precondition	Search input is not empty
Postcondition	Users are presented with the articles related to the keyword they searched on. Keyword may appear in any part of the article

| Use Case 3.1.5	An advanced search bar that allows input of individual criterias of article ✅
Users	Frontend Platform Users - General Population and EpiWatch Team
Overview	Upon clicking the Advanced Search Button on the Search Page, a side panel will open with multiple criterias and its input. Users can input the criteria they wish to search on into the fields and click search at the bottom.
Category	Phase 2 Frontend - Search Page
Trigger	User
Precondition	Side Panel is Active, Required Fields are not empty
Postcondition	Users are presented with the articles related to the keyword they searched on. Search Results are based on the combination of the criterias in the advanced search fields.

Use Case 3.1.6	Link to SEIR Model on CDC Articles ✅
Users	Frontend Platform Users - General Population and EpiWatch Team
Overview	Upon clicking on CDC articles sourced from our own api, an option "Model" will appear and clicking it will link to the SEIR model with that data of the specific disease.
Category	Phase 2 Frontend - Search Page
Trigger	User
Precondition	Article is sourced from our SENG3011_medics api
Postcondition	Users are redirected to the SEIR model tab

3.2 Statistical Analysis for Prediction

Use Case 1.2.1	Prediction Page - Implemented with SIR epidemiology mathematical model ✅
Users	Frontend Platform Users - General Population and EpiWatch Team
Overview	Upon selecting an outbreak on the Prediction Page, the graph will update with the number of cases / deaths for the specific selected outbreak. The graph will show the curve of cases and utilising the SEIR framework, predict the number of cases in the future. Variables that affect the infection curve are adjustable on this page.
Category	Phase 2 Frontend - Prediction Page
Trigger	Selecting an outbreak from dropdown
Precondition	There are at least 1 outbreak in database
Postcondition	Graph to show the predictive line of the number of cases

Use Case 1.2.2	Prediction Page - Save / delete different views to easily jump from model to model ✅
Users	Frontend Platform Users - General Population and EpiWatch Team
Overview	Upon adjusting the infection curve by changing variables, user can also save their current view as a preset. This preset can be accessible and upon clicking the preset, the curve variables return to its saved state. There must also be a delete button that will remove the preset from the application.
Category	Phase 2 Frontend - Prediction Page
Trigger	Save button on preset
Precondition	Curve dataset is not empty
Postcondition	A preset that can be used repetitively by clicking into it

Technical Architecture
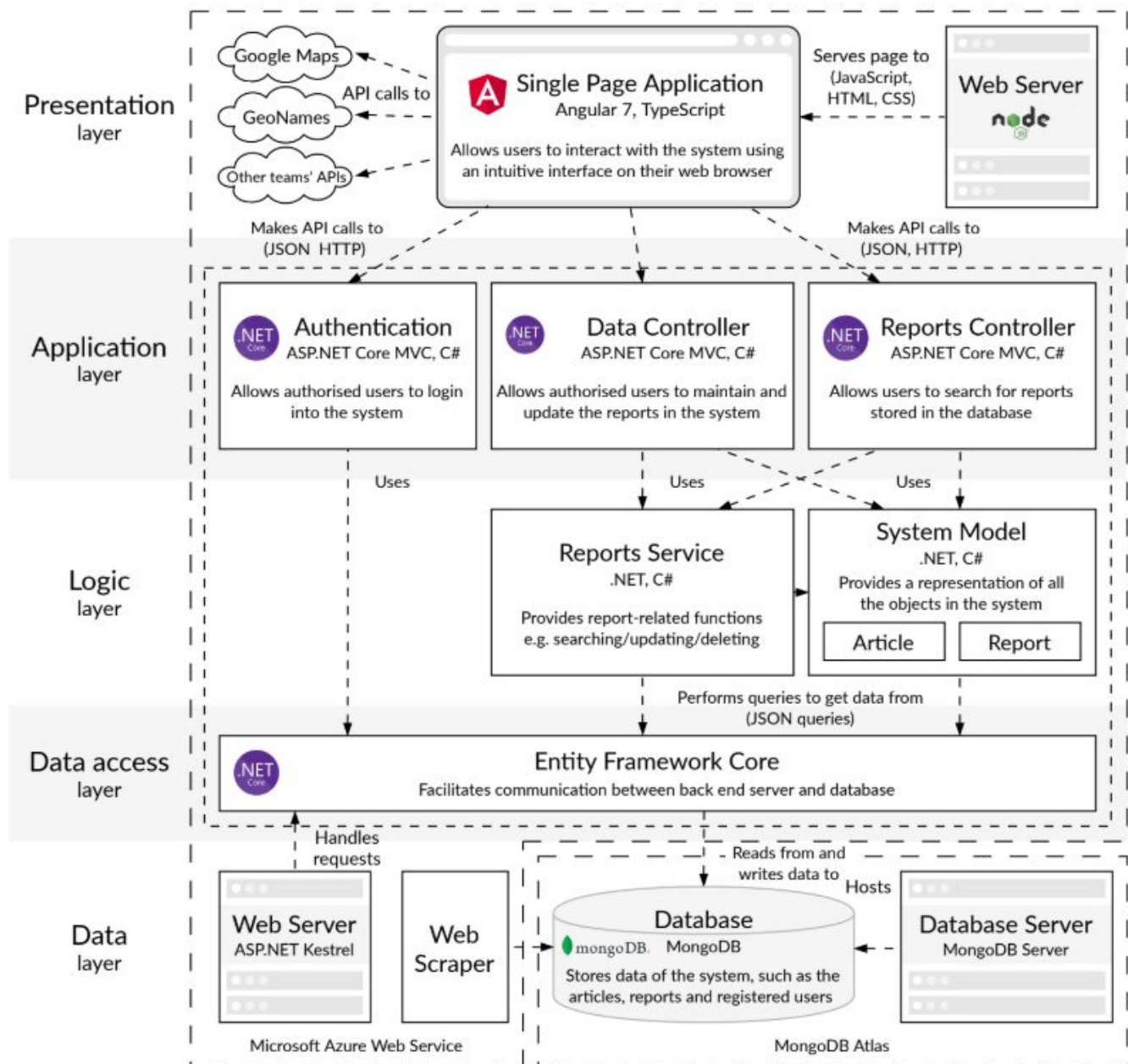
Integration with our API

Our platform was thoroughly integrated with our API, as we used it for both our map and model pages.

On our map page, which displays articles relevant to the given search criteria on a map, we used our API along with several other groups' APIs to retrieve relevant articles. We requested articles from these APIs by calling the *fetch* method in the base ArticleApi class (in src/app/apis/articles/base/ArticleApi.ts). We created subclasses of the ArticleApi class to handle the differences between the different teams' APIs, from creating the query strings to handling the returned object.

On our model page, which allows users to view and experiment with the parameters of an SEIR model for a particular disease, we used our API to extract case numbers for certain diseases. These case numbers were used to generate the model.

Data Conversion

Any time we wanted to make an API call to retrieve articles, we needed to convert our inputs into an appropriate form. For our API (and most other APIs), this meant:

- Converting the start and end dates, which were represented by JavaScript Date objects in our front-end components, to strings with the format "yyyy-MM-ddTHH:mm:ss".
- Converting the list of key terms, which was represented by an array of strings in our front-end components, to a single comma-separated string.

These conversions were performed by code in src/app/apis/articles/base/ArticleApi.ts, which meant that our front-end components did not need to be concerned with performing conversions.

Mapping from Retrieved Data to Visualised Data

API Responses

We needed to convert the responses from each API into a standard form that our front-end components could easily work with. This is because there were many subtle differences in how teams represented articles, reports, and locations, and how their APIs responded. For example:

- The simplest API response would be an array of article objects. A few APIs gave this response, but many other APIs bundled this array with other information (such as metadata) into a larger object. The difference is shown in the panel below.

```
// array of article objects
[
    { article object },
    { article object },
    ...
```

```
]

// object including other information
{
    "articles": [
        { article object },
        { article object },
        ...
    ],
    "metadata": {
        ...
    },
    ...
}
```

- There were three different valid ways to represent a location: (1) as a combination of country and sub-location, (2) as a GeoNames ID, and (3) as a Google Places ID. This implied that we would need a standard location object that would be capable of handling all three representations.
- Some APIs included additional information on top of the information required by the specification, such as article IDs (to be used in future calls to the same API) and case counts.
- Some teams gave their object fields different names from the names recommended in the specification - for example, "pubDate" instead of "date_of_publication".
- Some teams used different date formats from the format recommended in the specification - for example, "YYYY-MM-DD" instead of "yyyy-MM-dd HH:mm:ss".

Our solution was to create classes to represent standard articles, reports, and locations (in src/app/types/), and convert the objects in all API responses to these classes. We performed the conversion in the classes responsible for handling the subtleties of the different APIs (in src/app/apis/articles/concrete/). This conversion was essential - without it, our front-end components would need to handle many different object schemas that represent the same thing (i.e., articles, reports, and locations), which would lead to extremely disorganised code. With the conversion, our front-end components were allowed to assume that all articles returned from the ArticleRequest class were in that standard form, and could easily perform further transformations on the data if necessary. Our standard objects also enabled us to work on separate parts of the code without interfering with each other.

Synonyms

We needed to handle the fact that there could be multiple names for the same disease, and that APIs could return any of these names. For example, one API might use the name "coronavirus", while another API might use the name "COVID-19", but both names refer to the same disease. We dealt with this issue by normalising disease names (in src/app/apis/articles/base/ArticleApi.ts) before the articles were returned to the front-end components. This ensured that there would be only one name for each disease in the front-end.

Formatting

We also needed to perform some conversion for formatting purposes. Many of the date strings in the article and report objects returned by most APIs contained x's in place of unknown date components, as this was recommended by the specification. Displaying these raw strings to the user is obviously not desirable, as the x's would be confusing, so we needed to convert these strings to proper dates. We achieved this by converting each date string into a pair of dates which are the earliest and latest dates that could be interpreted from the string. For example, "2020-01-01 xx:xx:xx" was converted into the pair of dates "2020-01-01 00:00:00" and "2020-01-01 23:59:59". For the publication date, we simply took the earlier of the two dates, as the exact time was not important.

Obtaining Map Coordinates

Since one of our requirements was to display articles on a map, and almost no APIs provided coordinates in their location objects, we needed to obtain most of the coordinates by ourselves. We discovered two useful APIs for this: Open Street Maps (OSM) API and GeoNames API. These APIs are discussed in the "Additional APIs" section. After obtaining the coordinates, we were able to generate markers on the map for each unique set of coordinates and associate a set of articles with each marker.

Main Text Preview

The main text of most of the articles we retrieved contained hundreds of words, which was too much to be displayed to the user. Thus, instead of displaying the entire main text, we generated a preview of the main text, which went up to the end of the first sentence containing the 75th word, as we found that the first 75 words would typically contain enough information to convey the main point of the article. We then gave the user the option to open the original article if they wanted to see more.

Additional APIs

Our platform used a number of additional APIs to provide functionality that we would not have been able to provide otherwise.

Location APIs

Since one of our requirements was to display articles on an interactive map, we needed to be able to convert locations to a latitude and longitude.

To acquire this functionality we made use of the Open Street Maps (OSM) API and the GeoNames API. The OSM API was used in situations where only the country and city names were known, and the GeoNames API was used in situations where the GeoName ID was known. By using a combination of these APIs we were able to retrieve the latitude and longitude for every article. We chose to use these APIs over the Geocoding API as the Geocoding API is a paid API service, while the OSM and GeoNames APIs are free alternatives that run on an hourly credit system.

The logic for invoking these APIs based on the location information provided was handled inside a location mapper service (in src/Services/location-mapper.service.ts). The service would invoke the respective API based on the location information provided and return a latitude-longitude pair for use by the map component.

Map API

Our map was created using Google Maps API. Google offers an AngularJS package known as "AgmMap" which allows us to display an interactive map on the front end and offers customisation features for programmers. We are able to add custom markers to the map by providing the map with a list of marker objects. Marker objects simply contain the latitude and longitude of the marker and other visual settings. AgmMap also provided us with info windows, which we used to display a list of articles whenever a user clicks on a marker.

Other Teams' APIs

To obtain a wider variety of articles (from more locations, as the CDC website is primarily US based), we attempted to use other teams' APIs. We aimed to get at least one API for each data source (e.g., WHO, CIDRAP, Flu Tracker, etc.). Thus, we wrote code to fetch articles from these teams' APIs:

- ApiDemic (H5N1)
- CalmClams (Global Incident Map)
- Codeonavirus (WHO)
- ExceptionalCoders (Flu Tracker)
- FlyingSplaucers (CDC)
- Fm (ProMed)
- GnarlyNarwhals (CIDRAP)
- Webbscrapers (WHO)

The code for handling each of these APIs is in src/app/apis/articles/concrete/. Although we were able to successfully call each of these APIs when testing the articleStore module in node, we were not able to retrieve articles from many of these APIs when running the app in the browser. This was because many teams had not enabled CORS on their API. Thus, in the end, we were only able to reliably fetch articles from a few groups - ExceptionalCoders, Fm, and Webbscrapers.

Additional Libraries

SEIR

To increase the accuracy of our modelling page and minimise development time, we chose to rely on the SEIR Node.js package (https://libraries.io/npm/seir) after some browsing. It involves solving a system of four deterministic differential equations comprised of up to 18 parameters, and generates an estimated timeline of the modelled disease.

Storing Locations

The free location APIs that we used (OSM API and GeoNames API) had a major drawback, in that only 1000 requests were allowed per hour. During development, we would reach this limit fairly quickly, as we were constantly refreshing the browser while testing our application. This became a huge problem as after we had used up all 1000 requests, there would be no way to obtain the latitude and longitude of any locations until the next hour, resulting in an empty map.

One approach that we used to reduce the number of requests was to store the dictionary which contained the data model of our map into session storage. This meant that on refresh, the application could load the map instantly from session storage instead of calling the location APIs and wasting credits. Whilst this approach did work when we were simply adjusting and testing the appearance of our UI, it did not help when we were testing our search functionality, as the application would need to obtain coordinates for the locations in the articles returned by our (and other teams') APIs.

Our final approach was to create a new API endpoint that enabled us to store the result of calling the OSM and GeoNames APIs. Before calling the OSM or GeoNames API to obtain the coordinates for a location, we would first call our own API to check whether the coordinates were stored in our own database. If they were, then we could use these coordinates, saving us one request. Otherwise, we would call the OSM or GeoNames API as usual and store the result into our own database for future use. This greatly helped us to save credits and increased the speed of our application, as it now required far fewer external API calls.

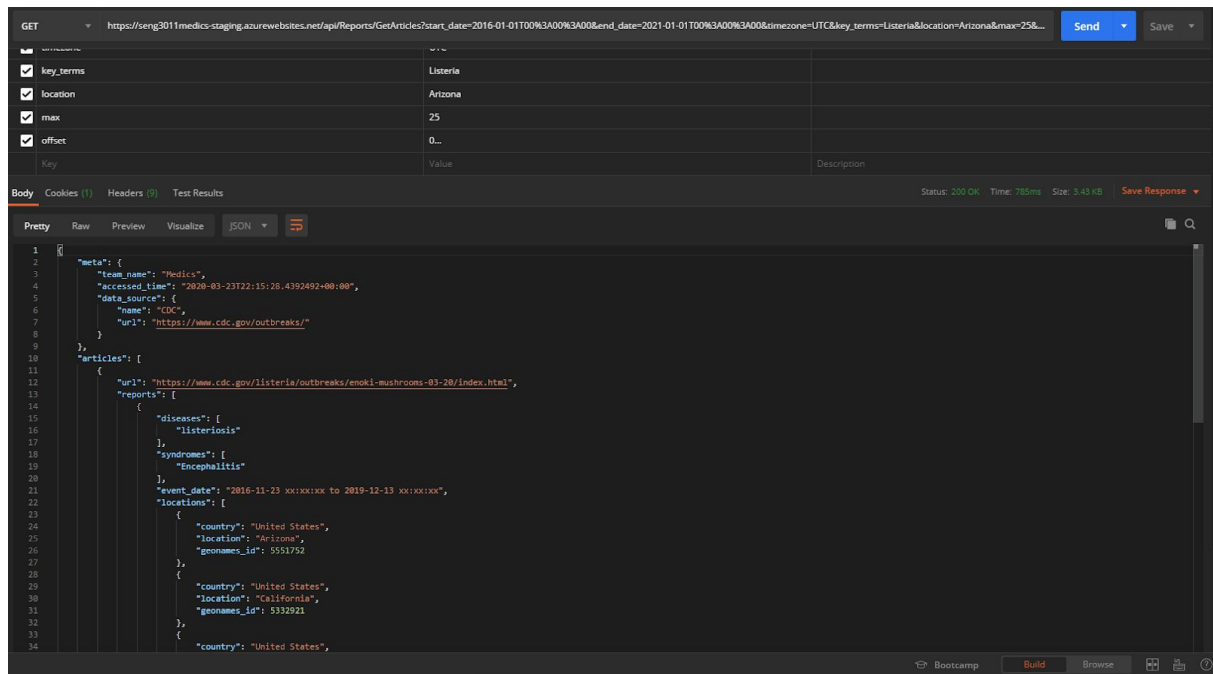# Appendices

Appendix A

Snapshots of API response logs

Figure 1: Postman request for logging example.

and laboratory evidence indicates that enoki mushrooms labeled as â€œProduct of Koreaâ€ are the likely source of this outbreak. S tate and local public health officials interviewed ill people about the foods they ate in the month before they became ill. Twelve out of 22 (55%) reported eating mushrooms, including enoki, portobello, white, button, cremini, wood ear, maitake, and oyster. Mic higan Department of Agriculture and Rural Development collected mushrooms for testing from a grocery store where an ill person pur chased enoki mushrooms. Two samples of enoki mushrooms yielded the outbreak strain of Listeria monocytogenes. These mushrooms are labeled as â€œProduct of Koreaâ€ and were distributed by Sun Hong Foods, Inc. Additional product testing is ongoing in Californi a. On March 9, 2020, Sun Hong Foods, Inc. recalled enoki mushrooms (UPC 7 426852 625810) labeled as â€œProduct of Koreaâ€. Consu mers, food service operators, and retailers should not eat, serve, or sell recalled enoki mushrooms. Enoki mushrooms distributed b y Sun Hong Foods, Inc. do not account for all illnesses in this outbreak. FDA is working to identify the source of the enoki mushr ooms distributed by Sun Hong Foods, Inc. and determine if other distributors received the same enoki mushrooms. CDC is concerned t hat enoki mushrooms labeled as â€œProduct of Koreaâ€ may be contaminated with Listeria monocytogenes and are advising people at higher risk â€" pregnant women, adults ages 65 years or older, and people with weakened immune systems â€" to avoid eating any eno ki mushrooms labeled as â€œProduct of Koreaâ€, until investigators determine the source of contamination and if additional produ cts are linked to illness. CDC will provide updates when more information is available.\n\nFDA: Sun Hong Foods, Inc. Recalls Enoki Mushroom FDA: Outbreak Investigations of Listeria monocytogenes â€" Enoki Mushrooms Questions and Answers onÂ ListeriaPreventing Â ListeriaÂ infectionSolve Foodborne OutbreaksCDCâ€™s Food Safety Information\n\n","date_of_publication":"2020-03-01 17:40:00"}]

Total Time Taken For Request: 00:00:00.1033481

Figure 2: Log file after request was sent