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.

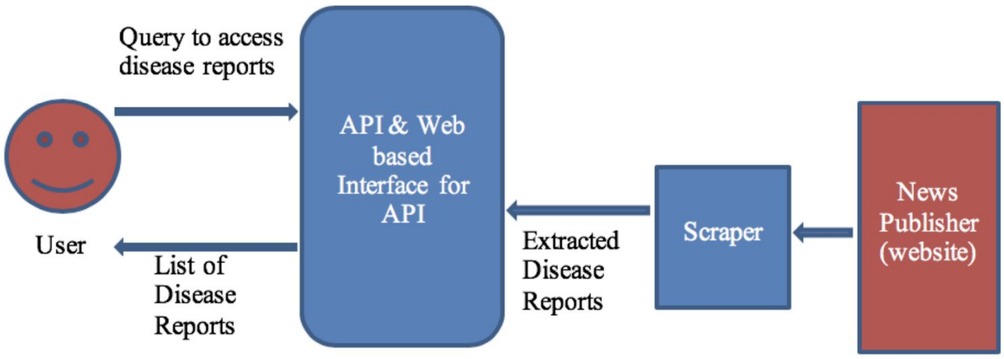


Figure 1: Desired Interaction between user and API endpoint.

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 disease reports. Currently, only one endpoint, /reports, is needed to achieve the functionality of accessing disease reports, but we may add more endpoints in the future to satisfy additional requirements.

While the API service is running, it will be connected to our reports database, hosted on MongoDB, where all the disease reports 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, is far too slow.

Testing

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

First, we will perform user acceptance testing. 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.

We will also be using Postman shown in Figure 2, a popular tool used in API testing, to test our endpoints locally before we push them out to production. Postman provides a clean interface that allows us to set up requests and see the response. Below is an example of a request and response made in Postman:

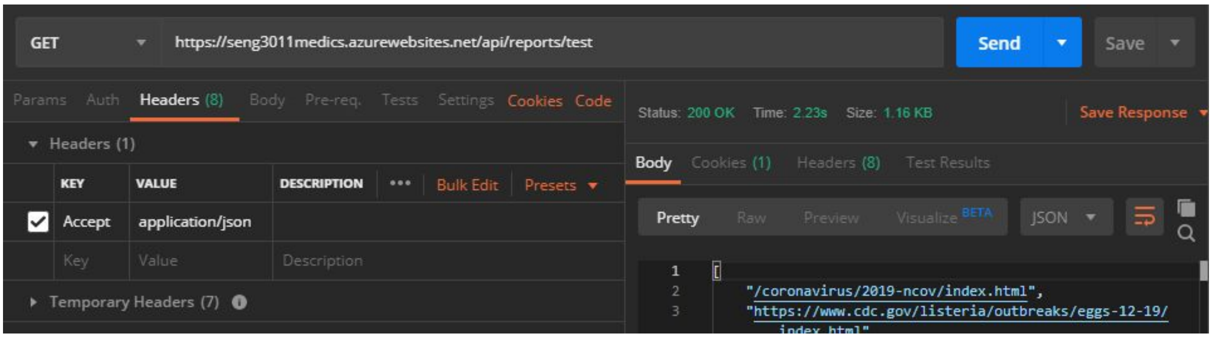


Figure 2: Example of request performed in Postman

We also plan to test our API by using unit tests. We will use a mock of the HTML client that will always return a faux outbreak page and outbreak article, from the CDC scrape. This will ensure that the unit test isn't dependent on dynamic data from a live web-page. Any service created will have a set of unit tests to ensure correct functionality.

Finally, we will 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

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).

Our API route will include a component for the version (/v1). This will be used to ensure that if we make a major change to our API (for example, changing how parameters are passed), clients using the original version of our API can continue to function. The new versions of the API would be accessible under /v2/reports, /v3/reports, etc.

Endpoint

```
https://seng3011medics.azurewebsites.net/api/v1/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

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

Description	In this interaction, the period of interest, 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%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”]
```

```
            }
        ]
    }
  ]
}
```

2. Search for all news of Anthrax in 2018:

Description: In this interaction, just the period of interest 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 period of interest 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. 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. In the below request, the user retrieves results 51-100.

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&location=China

Response	`400 Bad Request`
	
	`{`
	`    “error”: “No end date provided”`
	`}`

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=01-01-2017T00:00:00&end_date=12-31-2017T23:59:59&key_terms=Ebola&location=Egypt

Response	`400 Bad Request`
	
	`{`
	`    “error”: “Invalid date format”`
	`}`

9. Invalid date

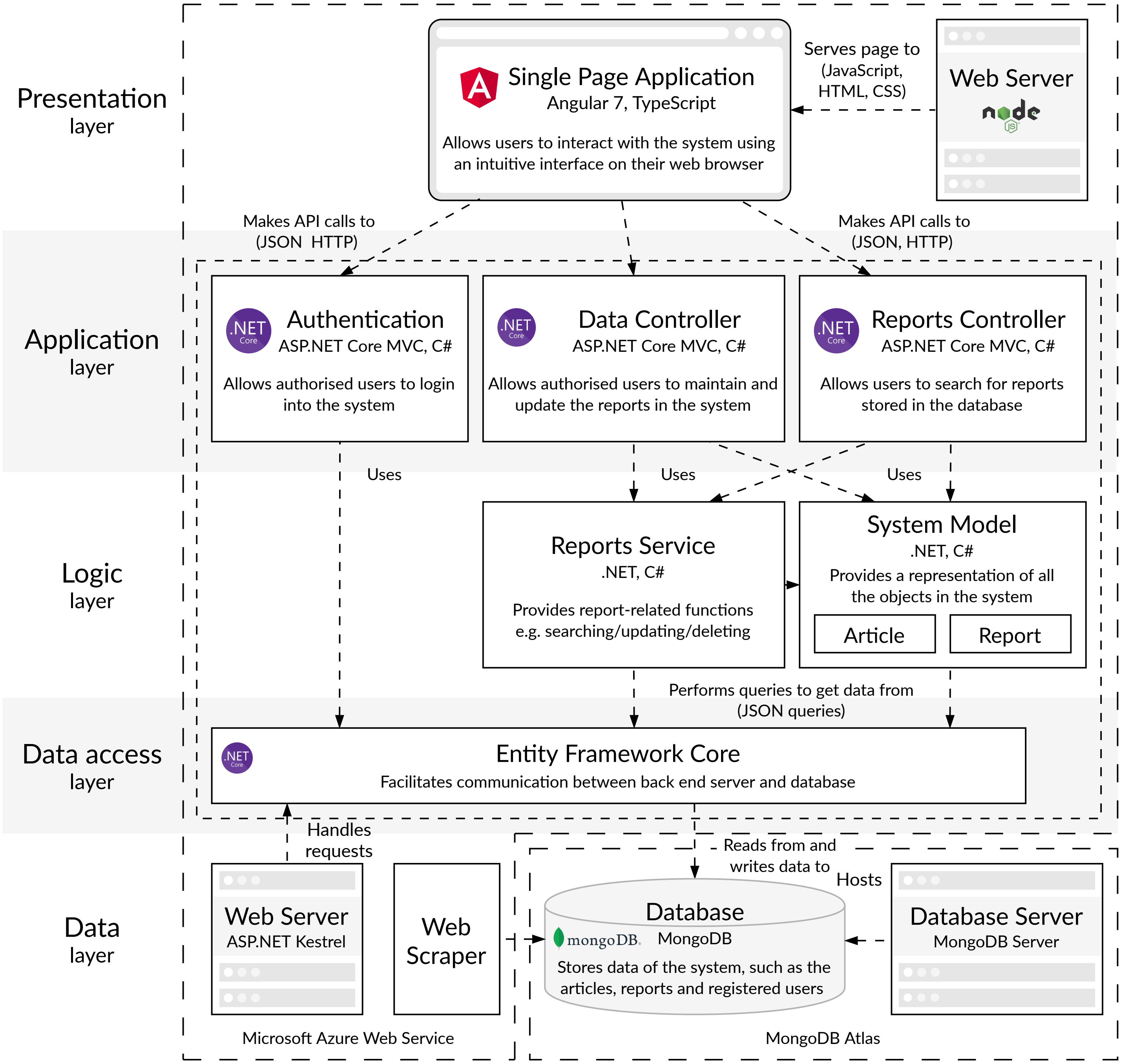
Description	If the provided date is invalid, an error will be returned.
Request	GET /reports?start_date=2019-01-01T00:00:00&end_date=2019-02-29T23:59:59&key_terms=Ebola&location=Democratic%20Republic%20of%20the%20Congo

Response

```
400 Bad Request

{
    “error”: “Invalid end date”
}
```

Software Architecture



Implementation

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.

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

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 data 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 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 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 (e.g., coronavirus instead of COVID-19), the scraper will need to convert disease names to a standard form.

After we have generated an article object for each article, the scraper will insert them into the database. To avoid unnecessary scraping, we will first check if the outbreak article exists in the database. If the outbreak 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.

The scraper will run periodically as a cron job to keep the data up to date with the CDC outbreak page.

Database Management System

We need a DBMS for storing disease 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, disease articles and reports are stored in JSON 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 setup and connect to a MongoDB Atlas cluster, and there is enough free storage to last us the duration of the project.

Front End Language

Our front-end will be implemented in AngularJS, as Angular is a JavaScript framework that componentises the frontend. 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 Angular 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. Angular 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 Angular, with the addition of performance, the reason we chose Angular over React was that Angular 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 do plan to use other packages for location parsing, 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 Break Down

Figure 3 displays the folder and class structure of our API.

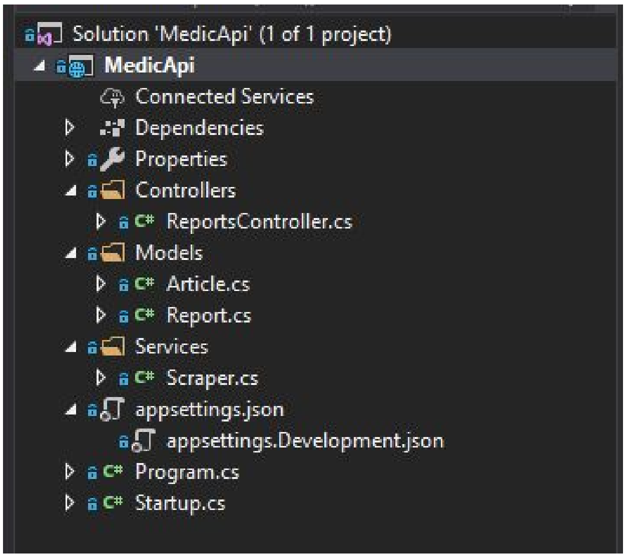


Figure 3: The Project Source Code File Structure

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.

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.

The Services folder contains code that handles all of the logic. This is to keep our controllers free of code and allow 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: Example of adding service as a singleton instance to startup

For controllers to access this service they simply 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

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.

Figure 7: example appsettings.json file used in 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.

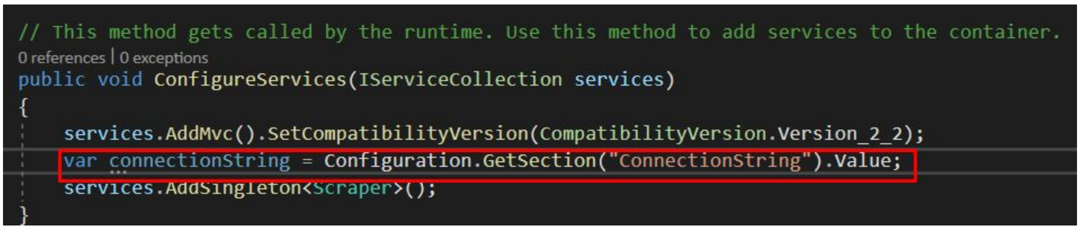


Figure 8: Binding settings from appsettings.json to variable example