Develop the API Module

Design and Implementation

We used REST (Representational State Transfer) architecture to build our API. REST is a style of architecture based on a set of principles (client/server, stateless, layered, and supports caching). In REST, the real world entity or an object is considered as a resource. Every resource is uniquely addressable using a uniform and minimal set of commands (typically using HTTP commands of GET, POST, PUT, or DELETE over the Internet). For this specific project, a resource is a report or an event. The architecture of REST is essentially the architecture of the Internet and that is why REST architecture is so popular.

Our API will serve as the service provider to provide disease reports to service consumers which are web applications or other APIs. The possible web applications that can be helped by our API include traveling applications, news applications etc. Basically, any application that needs to know about the risk of epidemic disease is our API's consumer.

For this project, we will have three independently working components, which are API, database and scraper. The scraper extracts data from the data source and stores the data in the database permanently twice a day. The API will get the data from the database and filter the data according to the clients' requests. Then, the API will create a dictionary as the response and send the response back to the clients in JSON format. This structure is decoupling and there are low dependencies between each component. So even if one component is broken or updated, the other components won't be affected much.

The inputs of the API are the period of interest (which is composed of "start_date" and "end_date"), "key_terms" and "location". The output of the API are disease reports satisfying those constraints.

Flask provides a route() decorator to handle the request for us. Then, we can use request.args function to get the parameters that the clients send to us. After getting all the parameters, we query the database using those parameters and extract data. In the end, we wrap the data in a dictionary, using the jsonify function to make a JSON file and sending the file back to the client.

Above is the flow of how every endpoint works. Since only dates are necessary and key_terms and location are not required, we came up with five endpoints.
/reports/<start_date>/<end_date>
/reports/<start_date>/<end_date>/<key_terms>
/reports/<start_date>/<end_date>/<location>
/reports/<start_date>/<end_date>/<key_terms>/<location>
/article/<article_id>
This is just our initial idea which may be changed during the process of implementation.

Documentation

Documentation is the key to a great experience when consuming our API. If we don't properly document our API, nobody is going to know how to use it, and it will be a horrible API. Good documentation is a concise reference manual containing all the information required to work with the API, with details about the functions, classes, return types, arguments and more, supported by tutorials and examples. To publish our API in public, we plan to use Swagger. Swagger is an open-source software framework backed by a large ecosystem of tools that helps developers design, build, document, and consume RESTful Web services. Also, a framework called flask_restplus can automatically generate Swagger UI documentation for us. Since we are using flask, this framework is suitable for us to generate documentation.

Testing

To do testing, we will use the software called Postman. Postman can act as a client when testing the application developed in Restful state. It allows us to see the response of a particular endpoint that the server returns.
Before we deploy the API to servers, we will use our local host as the server to run the API.

Authentication

We only allow registered users to use our API, so we can monitor consumption of the API and know who did what. So we will store the user information such as username and password in the database and when we will check the password when the user tries to use our API. In flask_jwt library, there is a class called JWT allowing us to implement the authentication feature.

Extension

For the extension, we plan to generate statistic figures like graphs along with the reports. Of course the user can choose to receive them or not. For example, if the user search for the disease Zika from 11th March to 15th March, our API will generate a list of statistic figures according to the data stored in the database and send those figures back to the client along with the reports.

Perhaps statistic figures, graphs? Or some summary of outbreaks happened in a particular location, in a period of time? Notifications for registered users?

Run it in Web service mode

For web service, we need server to handle requests and send responses back, mostly in json format. For other APIs, they would request our service for data. We then access and retrieve data from database and finally return that data as json.

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Our api will be run in web service mode as it is:

1. Available over the internet since we are deploying it to DigitalOcean.
2. Uses a standardized JSON messaging system so every consumer can access and understand the response.
3. Is not tied to any one operating system or programming language.
4. Is discoverable via a simple find mechanism.

The web service platform we are using is HTTP + JSON. The data returned by the API is encoded as JSON (Content-Type: application/json). The JSON will follow the format that's given in the project specification. The character encoding is UTF-8. We might provide the response in XML format if there is a huge demand in the feature (also as an extended feature).

HTTP status codes:
200 ---- Success
201 ---- Created
400 ---- Bad Request
404 ---- Not Found
500 ---- Internal Server Error

API calls

HTTP Verb	**URL**	**Response**

GET	/article/<articleID>	200
		Article found
		{
		    url: <string>,
		    date_of_publication: <string::date>,
		    headline: <string>,
		    main_text: <string>,
		    reports: [{
		        disease: [<string::disease>],
		        syndrome: [<string::syndrome>],
		        reported_events: [<object::event-report>],
		        comment: <string>
		    }]
		}
		
		400
		Invalid article ID

		404 Article not found
GET	/reports/<start_date>/<end_date>/<key_terms>/<location>	200 Search successful {[articles (see above for format)]} 400 Invalid parameters 404 No results found

Parameters

Parameter	**Required**	**Type**	**Description**
articleID	yes	Integer	ID of the article
start_date	yes	String in the form: “yyyy-MM-ddTHH:mm:ss”	Starting date and time of articles to be searched
end_date	yes	String in the form: “yyyy-MM-ddTHH:mm:ss”	Ending date and time of articles to be searched
key_terms	no	A list of string, with terms separated by commas	Comma separated list of key terms to be searched for
location	no	String	Name of the location interested in

The API is called using requests.get(), with the proper URL and a dictionary containing parameters.

For example, to perform a GET request to ‘hostname/reports’ (where hostname is the name of the API server), first a dictionary is created to hold the parameters:

```
        parameters = {
                                start_date: “2015-10-01T08:45:10”,
                                end_date: “2015-11-01T19:37:12”,
                                key_terms: “Anthrax,Zika”,
                                location: “Sydney”
                        }
```

Then the GET request is made:

```
        requests.get(“hostname/reports”, params=parameters)
```

The API extracts the parameters using requests.args.get().
For example, to extract start_date:

```
        start_date = request.args.get(‘start_date’)
```

Programming Language

For this project, we will be using Python in conjunction with the Flask framework. The two main languages we considered for developing the API and website were Python and NodeJS. Although NodeJS offers good performance, we believed that Python was a better choice for our group to use.

The main reason we decided on Python over NodeJS was comfort. Not all of the members in our group had experience using NodeJS, however we were all familiar with using Python. Writing code in Python is much easier and faster than in NodeJS, because Python's compact syntax requires fewer lines of code to achieve the same result. In addition, the larger variety of Python libraries facilitate the development of our API and website. For example, the BeautifulSoup library was used for scraping the CIDRAP website to extract the information our API will be accessing.

Development environment

The next decision was whether to use Django or Flask framework. Django has a lot of built in features, making it easy to start developing. Flask is more lightweight, but allows for greater flexibility. Also, Flask has many useful libraries for building the API. For example, flask has a library called flask_restful that's designed for building RESTful api. This is why we chose Flask.

Python libraries

Since we are using flask framework, we are going to use those libraries to build the api:
flask, flask_restful, flask_jwt, security, user

To make the scraper, we are going to use those libraries:
BeautifulSoup, requests, json

Database

We decide to use MySQL as our database for this project. The two options that we considered were MySQL and MongoDB. Although MongoDB stores data in JSON format and the output format of the API is defined as JSON format, MongoDB is more used in NodeJS programs while our team has decided to develop the backend in Python. Besides, since our team members all have experience using MySQL, it's easier for us to design and manage the database structure with it. We can store different aspects of the news such as time, location and disease type in the database by designing schemas for each of them. And we can then use commands like "SELECT', 'UPDATE', 'INSERT' and 'DELETE" to query the data and manage it. To get the required output format, we can transform the filtered data into JSON files before outputting. Therefore, MySQL is be a better choice for our project.

Deployment

After API implementation in backend, we decided to deploy the application on DigitalOcean Virtual Private Server running Linux system (Ubuntu 18.04) with all built-in dependencies required to run our application (Python, MySQL database, etc.). We can also register a domain name and have it point to the address of our VPS on DigitalOcean.

Reasons for choosing DigitalOcean VPS are:

- We get educational pack from Github for UNSW students credit
- Some of our members had experience running applications from DigitalOcean VPS
- DigitalOcean VPS provides many features, in which it can be run with dependencies and the environment of our choice, so we can run the application in the same environment that we implemented it in locally