# API Development & Implementation as a Web Service

We intend to develop our REST API module as a server-side *Web API*, that is, an interface consisting of one or more publicly exposed endpoints to a defined request-response message system which is exposed via a web server.

Our development process is structured as follows:

1. Building REST endpoints using Firestore's Cloud functions.
2. Routing the API URL to a Firebase HTTPS cloud function to map function name to URL name
3. Scraping and parsing data from our main data-source (flutracker.com) using Python's Beautiful Soup library
4. Converting this scraped data into JSON format
5. Adding a POST endpoint for ease of populating JSON data into Firestore
6. Setting up query parameters in our endpoints
7. Deploying our API using Firebase's CLI
8. Testing API endpoints and response format with Postman
9. Providing API documentation with Swagger
10. Adding in log files
11. Monitoring the performance of the API by in terms of response time by using "HTTPs network request trace" offered by Firebase
12. Allowing scope for scalability and further optimising response time by allowing caching of responses

This API serves as a medium to expose health and other disease related data to the browser. Once deployed and tested, our serverless API endpoints will be accessible as a web-service publicly to be used by other clients.

# Parameters and Responses

In accordance with standards for REST, we plan to pass input information to our module via HTTP GET requests. Parameters will be provided in the URL of our API endpoint in the form of query string parameters.
We expect the following compulsory parameters:

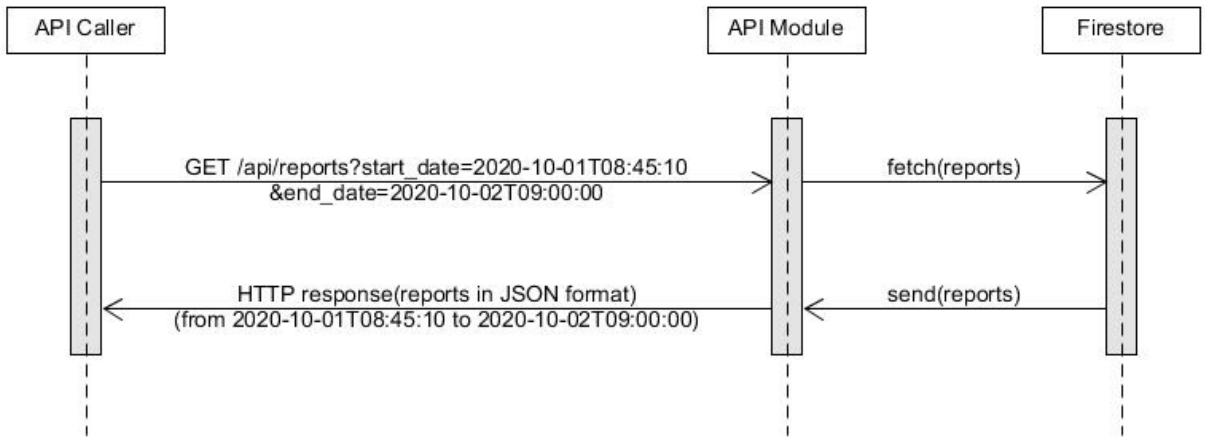
1. date::start_date - contains a single date (2018-xx-xx xx:xx:xx)
2. date::end_date - contains a single date (2018-xx-xx xx:xx:xx)
3. string::key terms - Comma separated key search terms. Note: key terms can be omitted.
4. string::location - contains a city/country/continent

And the following optional parameter:

1. string::timezone - string representing a valid timezone

Sample examples of our API endpoint structure:

1. GET /reports - to get a list most recent disease reports
2. GET /reports?start_date=2020-10-01T08:45:10&end_date=2020-10-02T09:00:00&keyterm=corona&keyterm=zika&location=sydney



Responses are collected in JSON format sent back in a HTTP response body.

Sample JSON Response:


```
{
	"url":"https://www.who.int/csr/don/17-january-2020-novel-coronavirus-japan
	-exchina/en/",
	"date_of_publication": "2020-10-01 13:00:00",
	"headline": "Novel Coronavirus - Japan (ex-China)",
	"main_text": "On 15 January 2020, the Ministry of Health, Labour and
Welfare, Japan (MHLW) reported an imported case of laboratory-confirmed
2019-novel coronavirus...",
	"reports": [
		{
			"event_date": "2020-01-03 xx:xx:xx to 2020-01-15",
			"locations": [
				{
					"country": "China",
					"location": "Wuhan, Hubei Province"
				},
				{
					"country": "Japan",
					"location": ""
				}
			],
			"diseases": [
				"2019-nCoV"
			],
			"syndromes": [
				"Fever of unknown Origin"
			]
		}
	]
}
```

# Technology Choices

(insert architectural stack)

## Comparison of backends

| | Google Firebase (Serverless) | Amazon EC2 (Serverless) | Server-based backends (e.g. Python + Flask / Django, LAMP, etc.) |
|---|---|---|---|
| Ease of Use | Very simple set up<br><br>The Firebase Admin SDK supports many languages including Python, Node, Java, Go, etc. to write scripts (serverless language abstracts away dependencies)<br><br>The SDK can also communicate between Cloud Functions (API endpoint) and the Firestore database | **Steeper learning curve**, infrastructure is geared more towards large teams<br><br>Has a number of individual modules (e.g. API Gateway, DynamoDB) required to connect all backend components<br><br>Client and server side code required for applications | Team has experience with Python + Flask / Django; Java + Spring; Node.js.<br><br>Has a number of individual modules to be coordinated |
| Performance | Usually low-latency (particularly real-time database), but serverless architecture means that there can be delays with “cold-start” - when service has not been “triggered” for a while | Excellent performance - high performing database in DynamoDB | No cloud and scalability means that the server can quickly become overloaded. |
| Interoperability | API endpoint be accessed by any HTTPS client<br><br>iOS, Android and JS clients share a Real-time | Host operating system is irrelevant as in serverless environments, there are no dedicated | Not cross-compatible on machines running different OS.<br><br>Individual frameworks |

| | | | |
|---|---|---|---|
| | DB<br><br>Host operating system is irrelevant as in serverless environments, there are no dedicated servers to service a given application, but rather a pool of OS agnostic resources (including servers, storage, etc.)<br><br>Reduces dependency on hardware and OS | servers to service a given application, but rather a pool of OS agnostic resources (including servers, storage, etc.)<br><br>Reduces dependency on hardware and OS | have dependencies |
| Scalability | Extremely scalable due to serverless architecture | Extremely scalable - Elastic allows changing capacity within minutes | Difficult to scale |
| Testing | Google allows remote testing from devices hosted by Google | AWS has support for many testing frameworks | Difficult to test - during development, server is hosted locally so difficult to stress-test |
| Data Accessibility | Data stored as JSON and every piece of data has a URL which can be used as a REST endpoint - allows viewing changes in real time | | |
| Documentation | Extensive documentation and video tutorials available | Good documentation | Extensive documentation |
| Community Support | Good community support | Sufficient community support | Sufficient community support |
| Pricing | Priced based on number of executions rather than pre-purchased compute capacity<br><br>Low IT costs | Pricing also based on executions rather than pre-purchased capacity<br><br>Slightly higher costs than Firebase - heavier set up | Pre-purchased capacity |
| Monitoring | App development | External monitoring | External monitoring |

| | | | |
|---|---|---|---|
| | platform includes crash analytics and performance monitoring | frameworks required e.g. Dynatrace, Datadog, Solarwinds | frameworks required |

Final justification for using firebase  + Also how using serverless means we don't actually need to make a choice on the deployment environment

The serverless architecture of Firebase was a key reason for selecting it. This produces great benefits in terms of cross-compatibility, scalability, price and not having to worry about a host operating system / deployment environment since these architectures use pools of OS agnostic resources. Uniquely as well, Firebase abstracts away most dependencies on backend hardware and servers, allowing us to be flexible with our implementation.

Firebase connects many components we require via the Admin SDK, allowing for simple coordination between the Cloud Functions API endpoint and the Cloud Firestore database, making for simple implementation for us. This was a major advantage over other serverless architectures such as Amazon EC2, which has many separated, more heavyweight components. Firebase also leverages Google's monitoring and analytics infrastructure and integrates this into its console.

Regarding criteria for which Firebase was not the favourite: although Amazon's backend services produce greater performance and allows quicker scaling, for the purposes of our Health API and the envisioned demand, Firebase suffices.

Firebase has an additional security advantage as it only accepts HTTPS requests, which provides the benefit of using Transport Layer Security to protect any transmitted data from tampering (say by a foreign actor trying to distort our epidemic statistics).

## Comparison of Web Scraping frameworks

| | **BeautifulSoup** | **Cheerio** | **Scrapy** |
|---|---|---|---|
| **Language** | Python | jQuery | Python |
| **Ease of use** | Easy to learn and use due to intuitive syntax and | Usage of jQuery increases learning curve | Substantial learning curve for beginners |

| | | | |
|---|---|---|---|
| | structure | | |
| **Documentation** | Extensive documentation | Decent documentation | Less than ideal documentation, especially for beginners |
| **Community Support** | Good community support | Community support is mostly for jQuery, not specific to cheerio | Good community support |
| **Performance** | Comparatively slow | Fast due to its simplicity | Fast as it uses asynchronous system calls |
| **Ecosystem** | Has lots of dependencies which is a downside | Confined to jQuery-specific libraries | Good ecosystem (especially for complex projects) |

We chose *BeautifulSoup* as our web scraping library due to its ease of use, extensive documentation and good community support.
While Cheerio and Scrapy had better performance in specific cases and a better ecosystem, they lack in their ease of use and learnability which is critical for our team and the project, especially considering the limited time-frame. Additionally, based on the difficulty involved in scraping from our website (FluTrackers) which is a series of forum posts directing to links from many external sources, it is important for us to use natural language processing libraries. Our team has experience using NLTK in Python, and so choosing to implement the scraper in Python is more convenient.

## Final API Design:

Our final API Structure has the following endpoints:

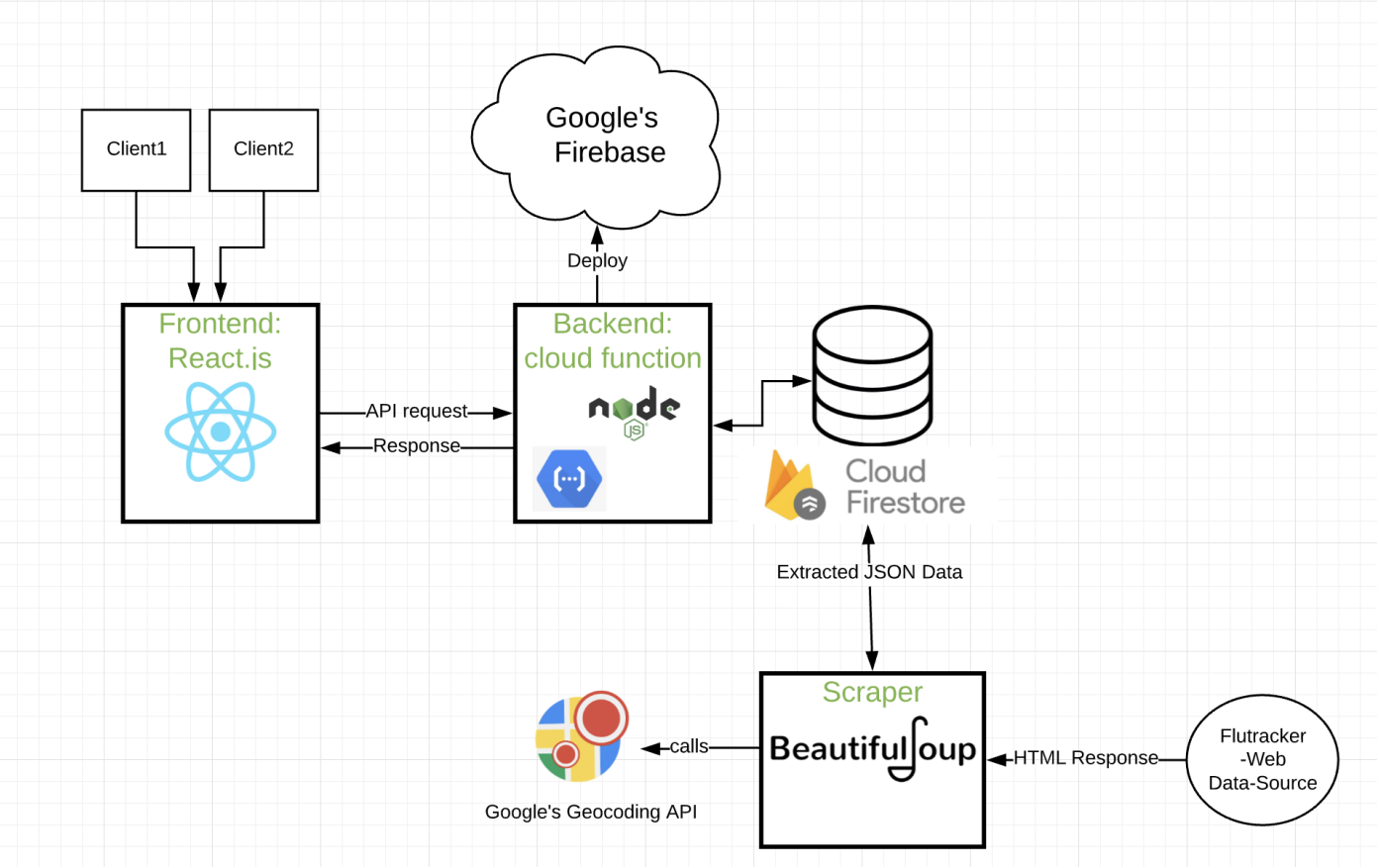
1. /articles - endpoint to retrieve specific articles
2. /logs - endpoint to retrieve logging history of all requests

**Changes in implementation and justification:**

Instead of using POST endpoints to populate our database, we decided to use Admin SDK in Firebase for the same.

Our main justification for doing so is:

- Direct communication with firestore to populate the database
- Concise code since we did not have to add an extra POST endpoint to push data
- Having a POST endpoint is a possible security vulnerability since it allows an external user to accidently or maliciously tamper with our data




Api architecture

## Challenges Addressed and Shortcomings:

The team addressed several challenges along the way. Below, we have grouped them into challenges faced with API design and challenges with scraping.

**API Design Challenges:**

1. Logging Data - Initially, we used “Morgan” a middleware node.js library to store all request logs in a ‘.log’ file since the spec mentioned we store log data into a file.

   Upon further discussion, we decided to use Firestore to store logs, and created a ‘/logs’ endpoint to retrieve logs as JSON objects.

2. Searching for articles (AND vs OR search) - In our first implementation of querying articles, we searched for articles that had:
   (Date between start-date and end-date) AND (the specified key-terms OR the specified location)

   For instance, searching with keyterm=corona and location=sydney would produce all articles with corona, irrespective of the location and also all articles that had sydney as location irrespective of the contents of the article

   However, on further discussion we decided that this was not the outcome we wanted. We re-implemented the logic as:
   (Date between start-date and end-date) AND (the specified key-terms AND the specified location)

   Thus, for the above search terms, only the articles featuring both corona and sydney would appear. When no key-terms or location are provided, all articles between the start and end date are produced.

   From a user’s perspective, this appeared more intuitive.

3. Date formatting - When retrieving articles from Firestore, the date_of_publication field in articles object was stored as a ‘timestamp JSON object’ (which is how Firestore internally stores Date() objects )

   We converted this JSON object back to a suitable DateTime object and parsed it to create our final date_of_publication field.

**Scraping Challenges:**

1. Inconsistent Data Source - The FluTracker website is extremely unstructured. For instance, it contains forum posts that do not contain information about outbreaks but has the keyterm “outbreak”

   Our scraper has to try and intelligently filter posts about outbreaks by making a guess based on whether the post contains identifying information such as locations, infection numbers and mortality numbers.

2. Variation in structure of sites that host articles: Since FluTracker allows users to post links to other external data sources, the articles containing the disease reports are usually not hosted by FluTracker, and so the pattern for scraping each article for its information is not consistent.

3. Location resolution: The types of location are not consistent, and sometimes, locations do not appear at all where they are expected to.

   Our scraper tries to search the headline for a location, and this can result in false positives. For example, a headline containing the word “us” might be incorrectly resolved to a location in the United States.

**Shortcomings:**

1. Hosting Location & Speed of Responses - Firebase’s Cloud functions have a limited choice of regions they can be hosted in. Our server is located in the US and thus, there can be a noticeable delay in the response after a request is made. A possible solution to this would be caching requests locally which the team plans to implement in later iterations of the project.

2. Requests/second vs responses/second - The API shows a linear increase in performance time as the number of requests sent increases. To improve on this, pagination of responses and adding a maximum limit to the number of documents served per user can be considered as a possible extension to the project that will be taken up in future iterations.

3. Firebase response size limitations: Since Firebase allows for a maximum of upto 10MB for HTTP responses, larger responses would not be received. Again, pagination is a possible solution to this.