Deliverable 1

Exception(al) Coders

Team Members

- Ravija Maheshwari (z5182793)
- Nithin Sudhir (z5113492)
- Saurabh Acharya (z5130134)
- Adi Ganguly (z5161646)

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
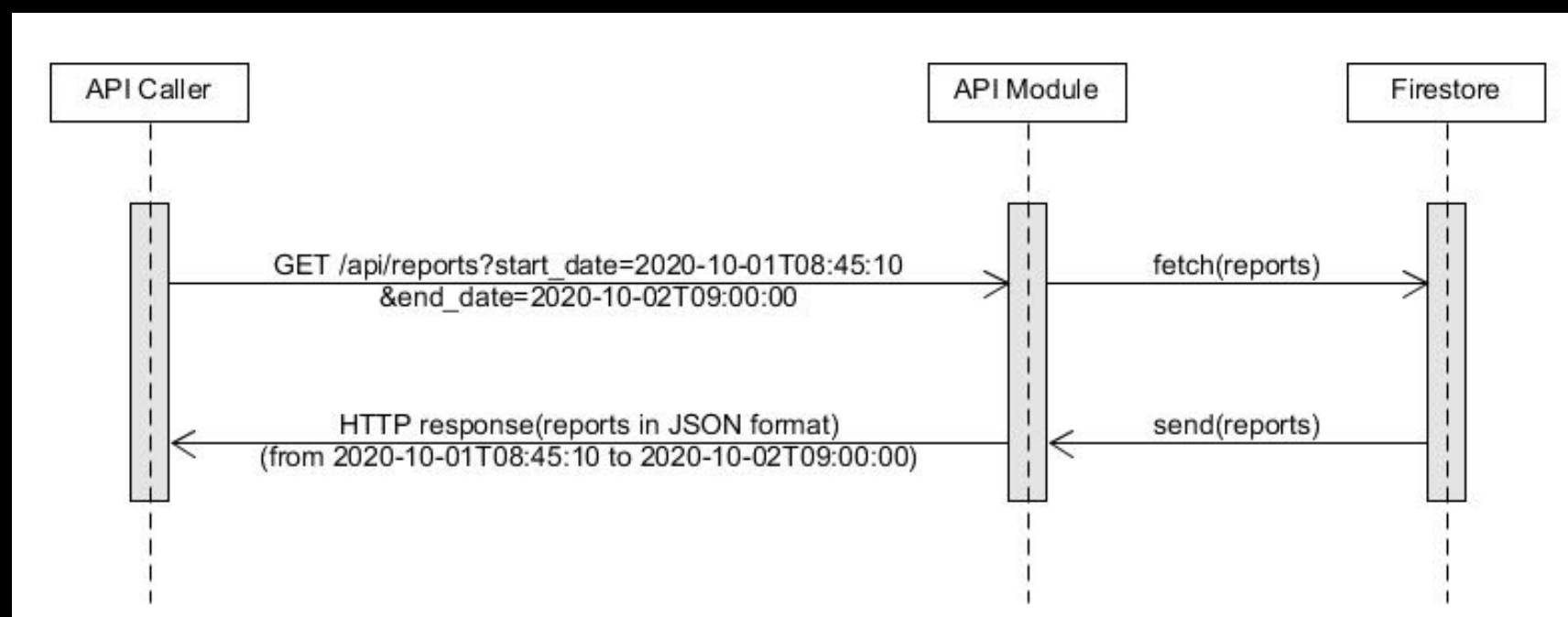
Sequence dig. for sample request and response

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

Sample JSON response:

```json
{
    "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

Comparison of backends:

	Google Firebase (serverless)	Amazon EC2 (serverless)	Server-based backends (Eg: Python+Flask/ Django, LAMP etc)
Ease of Use	Very simple set up The Firebase Admin SDK supports many languages including Python, Node, Java, Go, etc. to write scripts (serverless language abstracts away dependencies) The SDK can also communicate between Cloud Functions (API endpoint) and the Firestore database	**Steeper learning curve**, infrastructure is geared more towards large teams Has a number of individual modules (e.g. API Gateway, DynamoDB) required to connect all backend components Client and server side code required for applications	Team has experience with Python + Flask / Django; Java + Spring; Node.js. 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 iOS, Android and JS clients share a Real-time DB 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.) Reduces dependency on hardware and OS	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.) Reduces dependency on hardware and OS	Not cross-compatible on machines running different OS. Individual frameworks have dependencies
Scalability	Extremely scalable due to serverless architecture	Extremely scalable due to serverless architecture Extremely scalable - Elastic allows changing capacity within minutes	Difficult to scale

Note: Highlighted cells show most ideal options from each category

	Google Firebase (serverless)	Amazon EC2 (serverless)	Server-based backends (Eg: Python+Flask/ Django, LAMP etc)
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	Excellent performance - high performing database in DynamoDB	No cloud and scalability means that the server can quickly become overloaded.
Documentation	Extensive documentation and video tutorials available	Good documentation	Extensive documentation
Community Support	Good community support	Good community support	Good community support
Pricing	Pricing Priced based on number of executions rather than pre-purchased compute capacity Low IT costs	Pricing also based on executions rather than pre-purchased capacity Slightly higher costs than Firebase - heavier set up	Pricing also based on executions rather than pre-purchased capacity Slightly higher costs than Firebase - heavier set up Pre-purchased capacity
Monitoring	App development platform includes crash analytics and performance monitoring	External monitoring frameworks required e.g. Dynatrace, Datadog, Solarwinds	External monitoring frameworks required

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 structure	Usage of jQuery increases learning curve	Substantial learning curve for beginners
Documentation	Extensive documentation and online tutorials available	Good 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	Comparitively 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
- 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 have a steep learning curve which is critical for our team and the project, considering the limited 10 week 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.

Comparison of databases:

	Google's Firestore(noSQL)	MongoDB(noSQL)	MySQL (sql)
Primary DB Model	Document Store	Document Store	Relational DBMS
Languages	Firestore supports some popular programming languages (like Java, JavaScript, Python)	MongoDB has a much larger and exhaustive list of languages that it supports	MySQL supports a moderately large list of languages
Triggers	Allows usage of triggers with cloud functions.	Allows usage of triggers	Allows usage of triggers
Security	Less secure when compared to MongoDB	Provides good security of data	Not as good as non-relational databases
Scalability	Very Good scalability	Good scalability	Does not scale well

We chose Google's Firestore as our choice of database because it has a relatively easier setup process considering the team is using Google's firebase platform for the rest of the project.

Additionally, it is easier to store JSON data into a document store DBMS as compared to a Relational DBMS like mySQL.

Although MongoDB provides better security of data, for the purpose of this project the security provided by Firestore will suffice.

Package Management:

Dependencies with Node.js will be managed with npm, and the versions and packages will be defined in a *package.json* file, to ensure consistency during development and deployment.
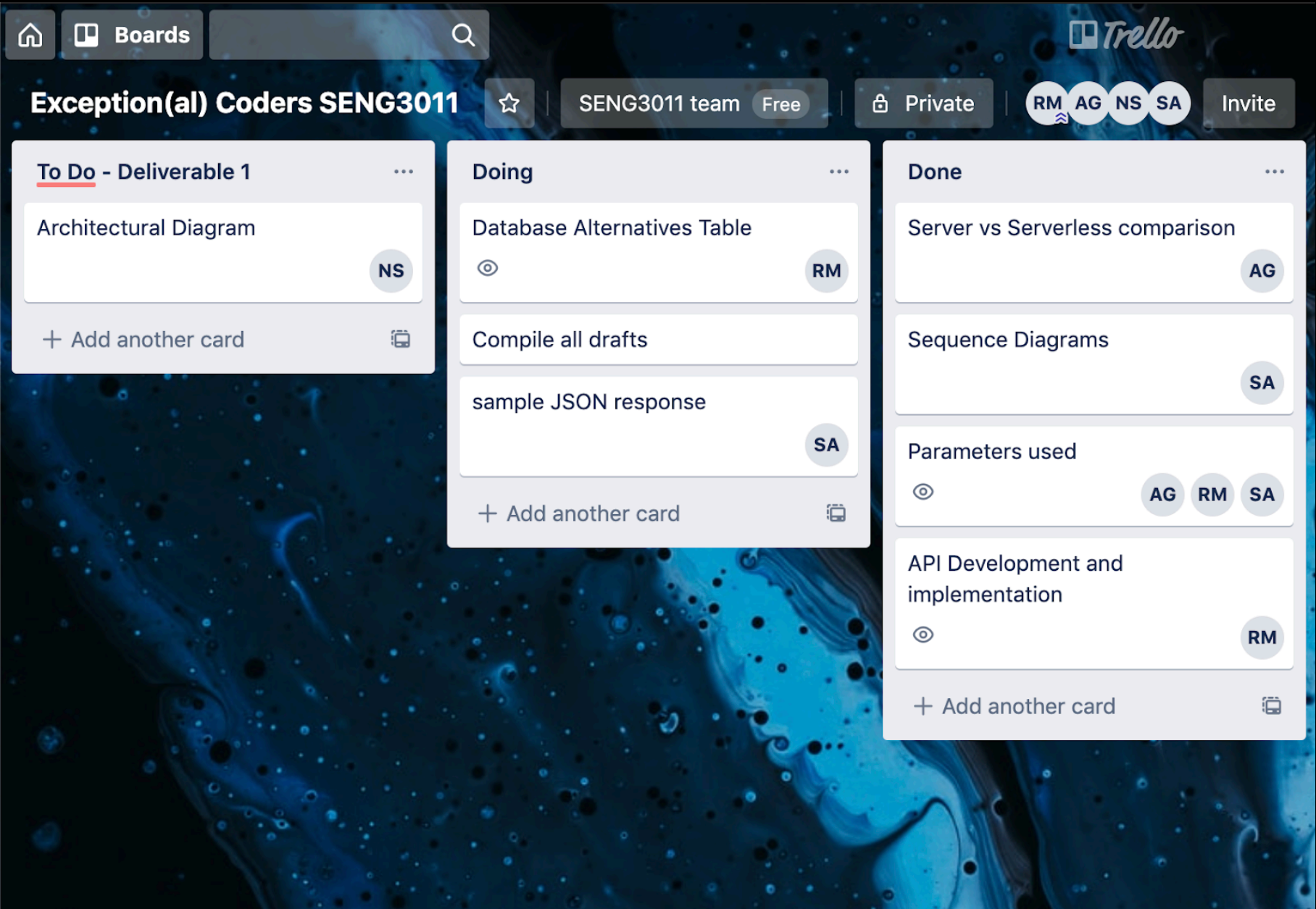
Other Libraries Used:

1. Natural Language Toolkit (NLTK) - for natural language processing and ease of parsing scraped date.

Project Management & Tools

Team meeting times: Monday 11:00 AM - 12:00 PM; Friday 10:00 AM - 12:00 PM.

The team is using a Trello board to track work division and coordinate activities in each deliverable.

Additionally, we are using a GANTT for work division between members.
The GANTT chart below indicates each member's approximate area of responsibility.
The key is as follows: Ravija (purple), Saurabh (blue), Adi (orange) Nithin (green)